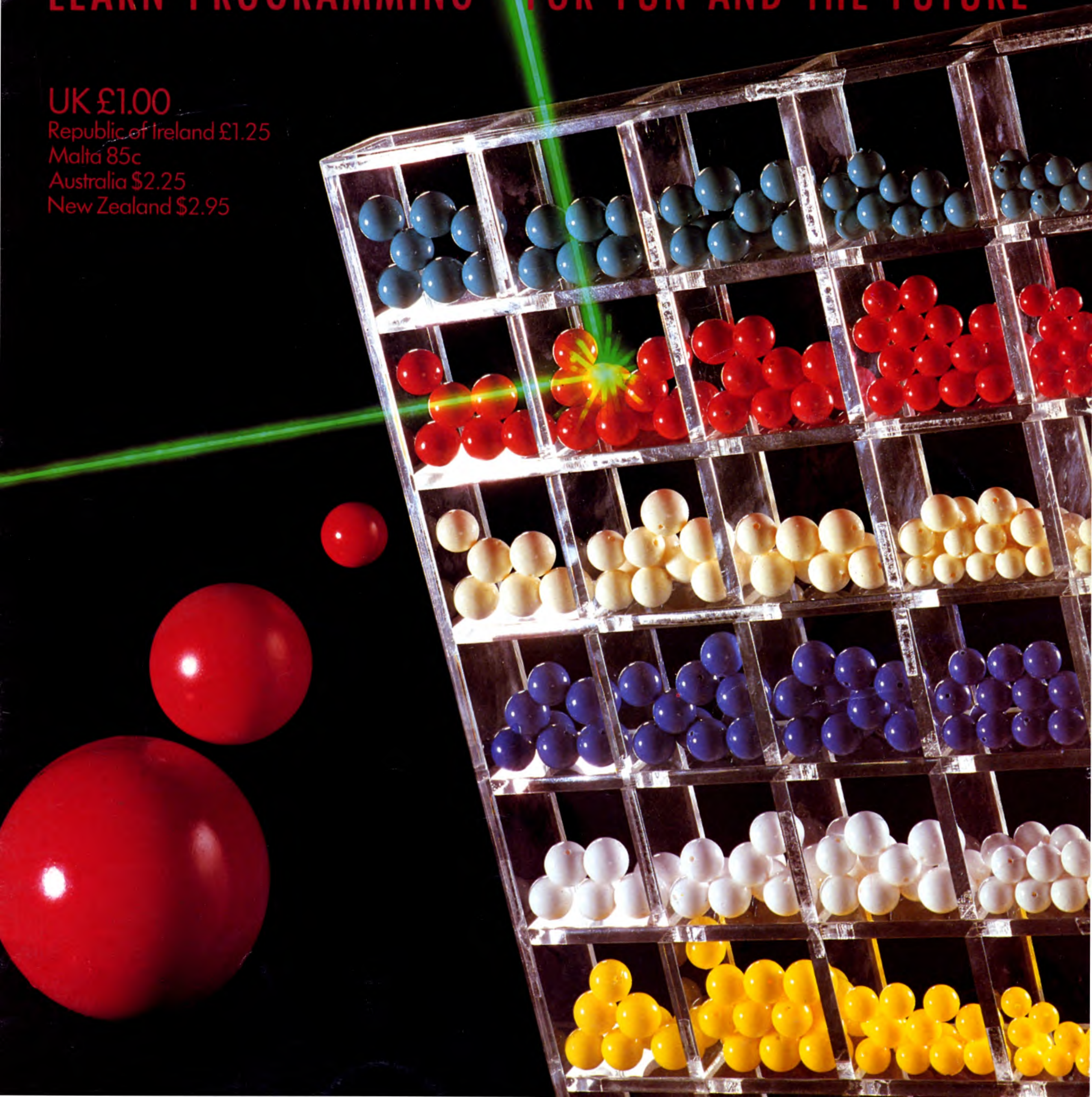
A MARSHALL CAVENDISH 9 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

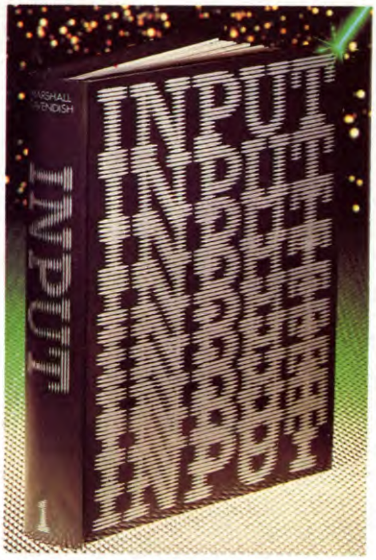
Vol 1 No 9

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front Cover, Dave King. Page 257, Nick Farmer. Page 258, K. Goebel/ZEFA/Hussein Hussein. Pages 260, 261, Ray Duns. Page 262, Tony Stone/Hussein Hussein. Pages 264, 266, 267, Alan Baker. Pages 269, 270, 272, Dave King. Pages 277, 279, Nick Farmer/Nick Mijnheer. Page 283, Digital Arts. Pages 284, 286, 288, Peter Bentley.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +

COMMODORE 64 and 128

ACORN ELECTRON, BBC B and B+

DRAGON 32 and 64

ZX81

VIC 20

TANDY TRS80 COLOUR COMPUTER

DISPLAY YOUR FACTS AND FIGURES

- ABOUT THE PROGRAM
- ENTERING THE INFORMATION
- EDITING THE DATA
- PLOTTING THE CHART
- SCALING THE AXES

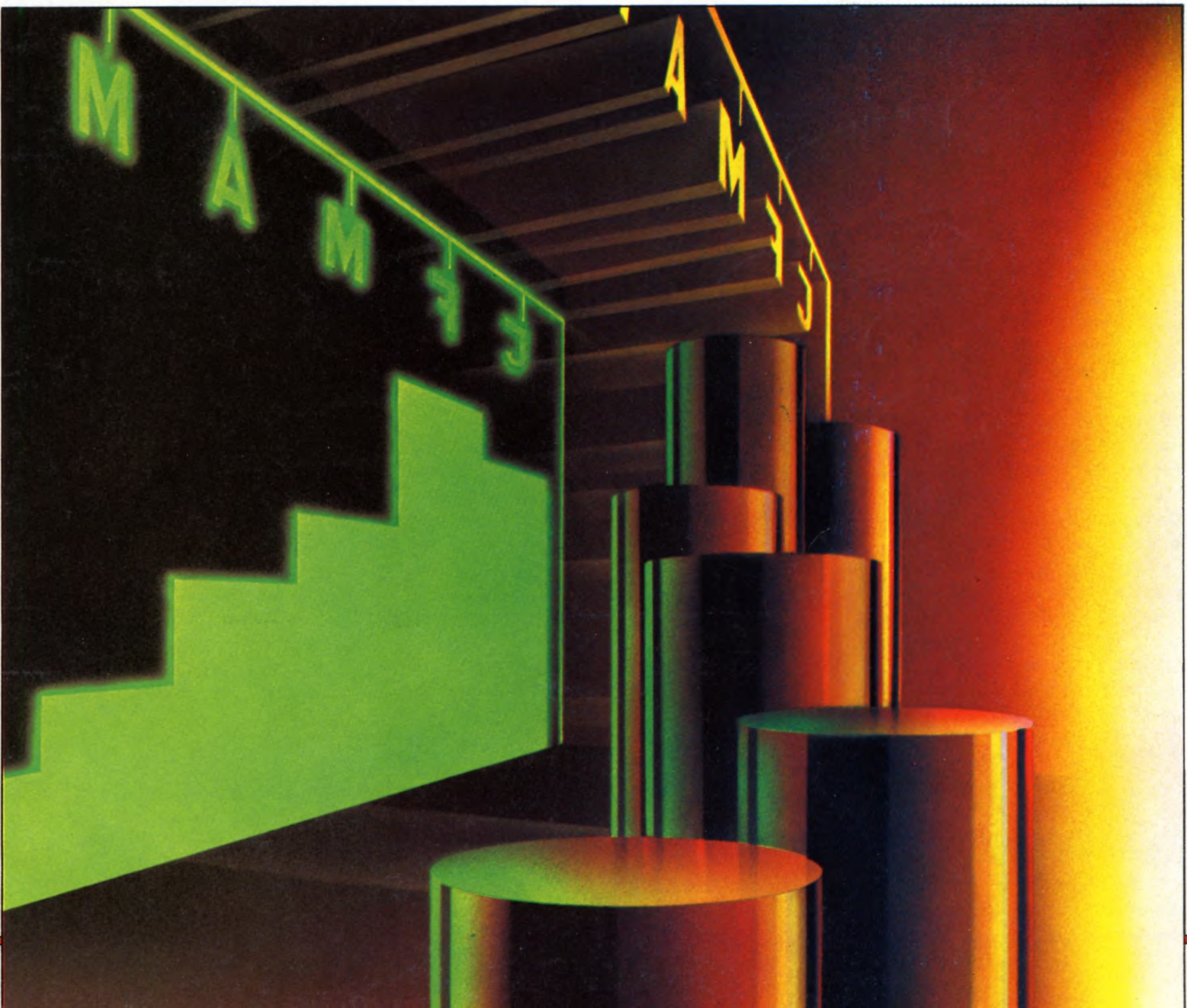
For instant analysis, you provide the figures, forget about the maths, and watch your micro convert them into a colourful and professional-looking bar chart

Everyone must have seen adverts for business computer systems showing elaborate displays of data, such as sales figures and stocklists. This kind of information—masses of virtually incomprehensible figures—is best presented as graphs or charts, which are easy to understand at a glance. Preparing them by hand is tedious, but it is just the kind of job at which business computers excel, with the ability to give a colourful display in seconds.

Data presentation isn't just limited to business machines, but it is also one of the practical applications for a home computer. All the machines covered here, with the exception of the Vic 20 and ZX81, have graphic and mathematical capabilities well able to handle this sort of work.

The average home user is unlikely to handle anything like the vast amount of information generated by even a small business, but there is much that can be analyzed usefully on a computer. For example, did income and expenditure increase towards the end of the year, or was expenditure falling? Did you spend more on computer software (or other items such as magazines, entertainment and motoring) during the autumn than you did during the summer? When did your savings rise above a certain level, and how long did they stay there?

Besides these commercial-type applications, there are all sorts of general interest



subjects, or perhaps facts and figures that relate to a hobby, which could usefully be analyzed and displayed. These subjects range from, for example, the attendance figures at a local club, to rainfall or tide levels. Other things you might want to chart are sports scores, and the size of a growing collection.

The simple program given here lets you rapidly prepare a visual display of any statistics that vary over a period of time. The axes of the display are adjusted automatically, so you can enter weekly, monthly or annual figures—in fact, any unit of time.

The maximum range of values that the program can cope with depends on the computer. The Dragon and Tandy are limited within +99 and −99. The Spectrum can handle up to a thousand, plus or minus, while the Acorn and Commodore computers handle numbers of any magnitude: units, tens, hundreds, thousands or even millions, if your accounts stretch that far.

USING THE PROGRAM

When you RUN the program, it displays a 'menu', or list of options. On selecting the option to enter new data, you are invited to name the axes—the labels you type in will appear on the chart when it is drawn. When entering the labels, remember to get them the right way round. The number of bars—representing weeks, months, years, or whatever—is plotted along the x-axis, and the value of the bars—dollars, pounds or any other units—is plotted along the y-axis.

Next, you are asked to enter the number of bars to be plotted. The maximum number you can enter depends on the arrangement of your computer—basically, the size of its graphics screen. For example, the Spectrum can plot a maximum of only 25 bars; the Dragon and Tandy plot 26; the Commodore 64 plots 30. Acorn machines are actually capable of plotting nearly 200, but the practical limit is about 80, beyond which the thickness of the bars is either irregular or too thin to be visible.

The display then asks you for your data. It prints out the number of each bar and asks you to enter its value, which can be either positive or negative. On the Dragon and Tandy, these values can range from −999 to 999; on the Spectrum they range from −1,000 to 1,000; on the Acorn and Commodore machines they range from -10^{38} to 10^{38}. If you use a Spectrum, Dragon or Tandy, and have data that lies outside the range of the program, then the solution is to plot the values in units of hundreds, thousands, millions, billions, or whatever.

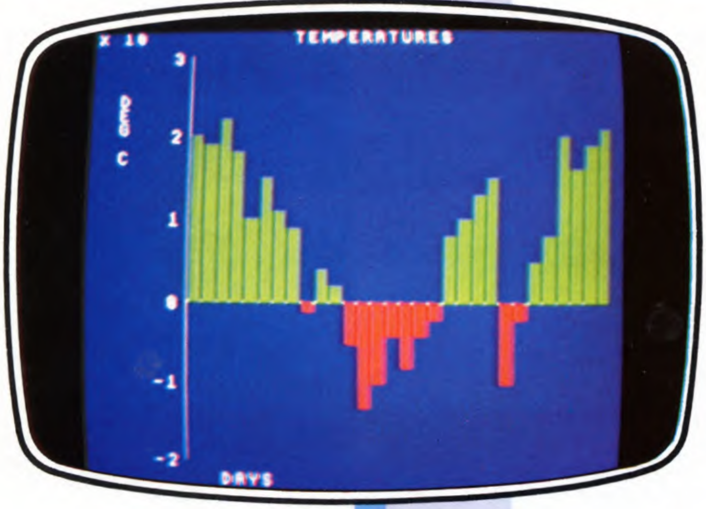
When the last value has been entered, the display returns to the menu, so you can choose either to edit the data—change the values you have entered, if you have made any mistakes—or plot the chart. If you choose to edit, the values will either be printed in turn on the screen (Commodore and Acorn), or you will be asked to enter the number of the bar you wish to edit (Spectrum, Dragon and Tandy). Commodore users should press any key but RETURN to leave a value unchanged and view the next one, or RETURN then enter the new value. Acorn users should press the space bar and Spectrum, Dragon and Tandy users should follow the prompts that appear on the screen.

When you are satisfied with the values, select the option to view or plot the chart. Acorn users have only one choice of chart, which is plotted almost instantly. On the BBC micro, the display shows the factor (for example, x1ØØ) by which the values along the y-axis are multiplied. This factor is printed on the first line above the y-axis, and on some TV sets may be partly obscured by the frame of the TV tube. To bring the factor fully into view, enter *TV255 RETURN before RUNning the program. The effect is to lower the display by one line.

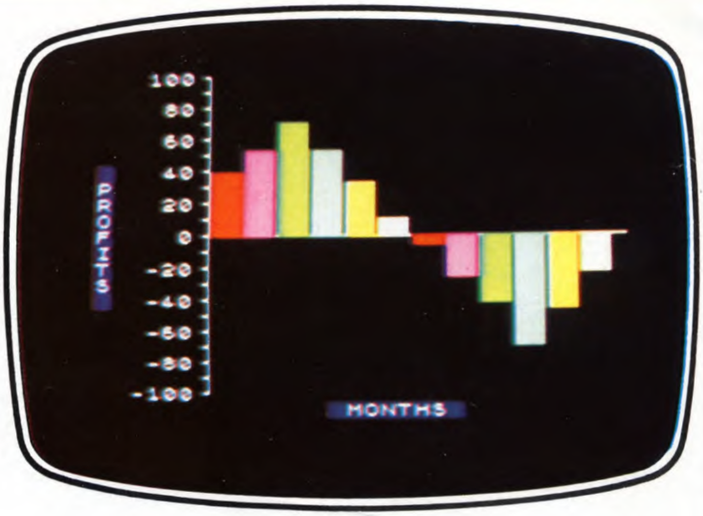
Users of Dragon, Tandy, Spectrum and Commodore micros can choose either a scaled graph or a full-scale graph. The scaled graph option displays the chart with dimensions along the y-axis rounded off to a maximum of ten, a hundred, a thousand or so on, depending on the maximum value of the data, but the chart might not fill the whole of the screen. The full-screen option displays the chart over the entire area of the TV screen, but prints the actual data values (not the scaled values) along the y-axis. The Acorn version always displays a full-screen graph.

For clarity, the bars plotted by the Acorn, Commodore and Spectrum micros are separated by a small space or coloured bar. This is not practicable on the Dragon and Tandy, so the bars are coloured alternately cyan and yellow for positive values, and red and orange for negative values.

```
10 LET g=Ø: POKE 236Ø9,2Ø: POKE
   23658,8
1ØØ BORDER 7: PAPER 7: INK Ø: CLS
11Ø PRINT BRIGHT 1; PAPER 3; INK 7;
   AT 4,9;"□O□P□T□I□O□N□S□"
12Ø PRINT BRIGHT 1;AT 7,6;"□1:—
   □ENTER□NEW□DATA□□□"
13Ø PRINT BRIGHT 1;AT 9,6;"□2:—
   □VIEW□/□EDIT□DATA□"
14Ø PRINT BRIGHT 1;AT 11,6;"□3:—
   □SCALED□GRAPH□□□□□"
145 PRINT BRIGHT 1;AT 13,6;"□4:—
   □FULL□SCREEN□GRAPH"
15Ø PRINT BRIGHT 1; FLASH 1; INK 2;
   AT 16,9;"□SELECT□OPTION□"
16Ø IF INKEY$="" THEN GOTO 16Ø
17Ø LET a$=INKEY$: IF a$<"1" OR
   a$>"4" THEN GOTO 16Ø
175 IF a$<>"1" AND g=Ø THEN GOTO
   16Ø
18Ø GOSUB VAL a$*1ØØØ: GOTO 1ØØ
5ØØ REM **NUMERIC INPUT ROUTINE**
51Ø INPUT (w$); LINE a$: IF LEN
   a$=Ø THEN GOTO 51Ø
52Ø FOR j=1 TO LEN a$
54Ø IF (a$(j)>="Ø" AND a$(j)<="9")
   OR a$(j)="." OR a$(j)="—" THEN
   NEXT j: LET v=VAL a$: RETURN
55Ø GOTO 51Ø
1ØØØ REM **INPUT ROUTINE**
1Ø1Ø BORDER 1: PAPER 1: INK 7: CLS
1Ø2Ø INPUT "NAME OF X—AXIS?□";
   LINE x$
1Ø3Ø PRINT INVERSE 1;AT Ø,Ø;"□";
   x$;"□"
1Ø4Ø INPUT "NAME OF Y—AXIS?□";
   LINE y$
1Ø5Ø PRINT INVERSE 1;AT Ø,16;"□";
   y$;"□"
1Ø6Ø LET w$="HOW MANY□"+x$
   +"□(1 TO 25)?□": GOSUB 5ØØ
1Ø7Ø IF v<1 OR v>25 OR v<>INT v
   THEN GOTO 1Ø6Ø
1Ø9Ø LET z=v: DIM a(z)
11ØØ FOR k=1 TO z
111Ø LET w$="ENTER DATA FOR□"+
   STR$ k+"□": GOSUB 5ØØ
112Ø LET a(k)=v
113Ø PRINT k,a(k)
114Ø NEXT k: LET g=1: PAUSE 5Ø:
   RETURN
2ØØØ REM **EDIT / VIEW ROUTINE**
2Ø1Ø BORDER 2: PAPER 2: INK 7
2Ø2Ø LET cn=1
2Ø25 CLS : PRINT PAPER 6; INK 2;
   AT Ø,Ø;x$,y$;TAB 31;"□"
2Ø3Ø PRINT cn,a(cn)
2Ø35 PRINT #1; PAPER 6; INK 2;AT
   Ø,Ø;"□□EDIT to alter current
   value□□□□□□any other key to
   continue□□□□"
2Ø4Ø PAUSE Ø
2Ø5Ø IF INKEY$="" THEN GOTO 2Ø5Ø
2Ø6Ø LET c$=INKEY$
2Ø7Ø IF c$=CHR$ 7 THEN GOSUB 25ØØ
2Ø8Ø IF cn=z THEN PRINT PAPER 6;
   INK 2;"END OF DATA": PAUSE 1ØØ:
   RETURN
2Ø9Ø LET cn=cn+1: IF cn=21 THEN
   GOTO 2Ø25
21ØØ GOTO 2Ø3Ø
25ØØ LET w$="ENTER NEW VALUE FOR□"
   +STR$ cn+"□": GOSUB 5ØØ
251Ø LET a(cn)=v: PRINT PAPER 6;
   INK 2;cn,a(cn);TAB 31;"□":
   RETURN
3ØØØ REM **SCALED GRAPH**
3Ø1Ø BORDER Ø: PAPER Ø: INK 7: CLS:
   LET hi=Ø: LET lo=Ø
3Ø2Ø FOR k=1 TO z
3Ø3Ø IF a(k)>hi THEN LET hi=a(k)
3Ø4Ø IF a(k)<lo THEN LET lo=a(k)
3Ø5Ø NEXT k
3Ø6Ø LET type=2: LET org=4
3Ø7Ø IF lo<Ø THEN LET type=1: LET
   org=84
3Ø8Ø LET h=hi: IF ABS lo>hi THEN
   LET h=ABS lo
3Ø9Ø LET ra=hi—lo
31ØØ IF h<=1 THEN LET hi=1: GOTO
   315Ø
311Ø IF h<=1Ø THEN LET hi=1Ø: GOTO
```



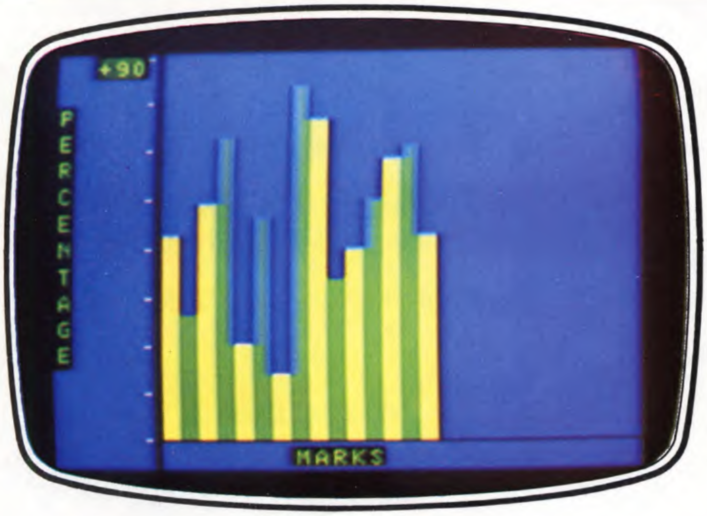
Acorn: a plot of seasonal temperature changes

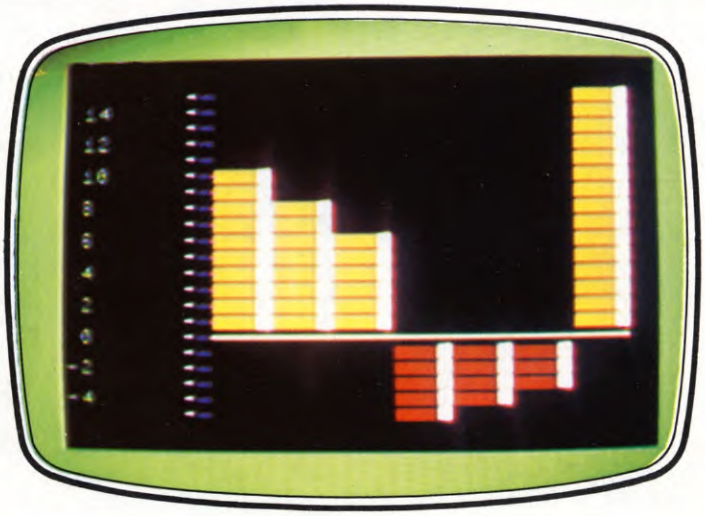


Spectrum: profits and losses for 12 months

```
    315Ø
312Ø IF h< =10Ø THEN LET hi=10Ø:
    GOTO 315Ø
313Ø IF h< =100Ø THEN LET hi=100Ø:
    GOTO 315Ø
314Ø IF h< =1000Ø THEN LET hi=1000Ø
315Ø LET wd=INT (25/z)
316Ø IF type=1 THEN LET ft=8Ø/hi
317Ø IF type=2 THEN LET ft=162/hi
318Ø PLOT 56,org: DRAW 198,Ø
319Ø PLOT 55,4: DRAW Ø,16Ø
320Ø FOR n=4 TO 168 STEP 8: PLOT
    52,n: DRAW 3,Ø: NEXT n
322Ø PRINT #1; PAPER 1;AT Ø,14;"□";
    x$;"□"
3225 LET z$="□"+y$+"□"
323Ø FOR n=1 TO LEN z$
324Ø PRINT AT n+(19-LEN y$)/2,Ø;
    PAPER 1;z$(n)
325Ø NEXT n
3255 LET dc=1
326Ø IF type=1 THEN GOTO 332Ø
327Ø FOR n=hi TO Ø STEP -(hi/1Ø)
3275 IF n<.Ø1 THEN LET n=Ø
328Ø LET n$=STR$ n
329Ø PRINT AT dc,(6-LEN n$);n
3295 LET dc=dc+2
330Ø NEXT n
331Ø GOTO 34ØØ
332Ø FOR n=hi TO -hi STEP -(hi/5)
333Ø IF n<.Ø1 AND n>-.Ø1 THEN
    LET n=Ø
334Ø LET n$=STR$ n
335Ø PRINT AT dc, (6-LEN n$);n
336Ø LET dc=dc+2
337Ø NEXT n
34ØØ LET ink=1
341Ø FOR n=1 TO z
342Ø LET cm=org
343Ø LET ink=ink+1: IF ink=8 THEN
    LET ink=2
344Ø INK ink
345Ø FOR m=1 TO (a(n)*ft) STEP SGN a(n)
346Ø PLOT 56+(n-1)*wd*8,cm: DRAW
    wd*8-2,Ø
347Ø LET cm=cm+SGN a(n)
348Ø NEXT m
349Ø NEXT n
35ØØ IF INKEY$< >"" THEN GOTO 35ØØ
351Ø IF INKEY$="" THEN GOTO 351Ø
352Ø RETURN
4ØØØ REM **FULL SCREEN GRAPH**
4Ø1Ø BORDER Ø: PAPER Ø: INK 7: CLS :
    LET hi=Ø: LET lo=Ø
4Ø2Ø FOR n=1 TO z
4Ø3Ø IF a(n)>hi THEN LET hi=a(n)
4Ø4Ø IF a(n)<lo THEN LET lo=a(n)
4Ø5Ø NEXT n
4Ø6Ø LET ra=hi-lo: LET ft=175/ra:
    LET org=(ra-hi)*ft
4Ø7Ø LET wd=INT (25/z)
4Ø8Ø PLOT 56,org: DRAW 198,Ø
4Ø9Ø PLOT 55,Ø: DRAW Ø,175
41ØØ PRINT #1; PAPER 1;AT Ø,14;"□";
    x$;"□"
411Ø LET z$="□"+y$+"□"
412Ø FOR n=1 TO LEN z$
413Ø PRINT AT n+(19-LEN y$)/2,Ø;
    PAPER 1;z$(n)
414Ø NEXT n
415Ø PRINT AT Ø,Ø;hi;AT 21,Ø;lo
42ØØ GOTO 34ØØ
```

```
Ø BB=5328Ø:GOTO 2ØØ
1 CLR:BB=5328Ø:FF=1
2 POKE BB,Ø:POKE BB+1,Ø
3 Q$="▣▣▣▣▣▣▣▣▣▣▣
  ▣▣▣▣▣▣▣▣▣▣▣▣▣
  ▣"
4 INPUT "♥ENTER□NAME□FOR□
  GRAPH";N$
5 INPUT "♥πENTER□NUMBER□OF□
  BARS. (1-3Ø)";XX
6 IF XX<1 OR XX>3Ø THEN 5
7 DIM A(XX),B(XX),C(XX)
8 FOR Z=1 TO 3Ø
9 IF XX*Z>3Ø THEN X1=Z-1: GOTO 11
1Ø NEXT Z
11 PRINT "♥ENTER□DATA:▣"
12 FOR Z=1 TO XX:PRINT "BAR"; Z;:
   INPUT C(Z):NEXT Z
13 A=Ø:B=Ø:C=Ø:D=Ø
14 FOR Z=1 TO XX
15 IF C(Z)>A THEN A=C(Z)
16 IF C(Z)<B THEN B=C(Z)
17 IF C(Z)<Ø THEN B(Z)=C(Z): A(Z)=Ø
18 IF C(Z)>Ø THEN A(Z)=C(Z): B(Z)=Ø
19 NEXT Z
2Ø C=A+ABS(B)
22 GOTO 2ØØ
24 POKE BB,5:POKE BB+1,Ø
25 PRINT "♥"N$
26 FOR Z=2 TO 22
27 POKE 1Ø33+Z*4Ø,64:POKE 553Ø5
   +Z*4Ø,6
28 POKE 1Ø32+Z*4Ø,45:POKE 553Ø4+
   Z*4Ø,3
29 NEXT Z
3Ø FOR Z=1 TO XX:W=Ø:CO=8
31 IF B(Z)<Ø THEN CO=2
32 FOR ZZ=B/D TO A/D
33 W=W+1:IF INT(ZZ)< >Ø THEN 38
34 FOR G=1 TO X1
35 POKE (1953-X1)+(Z*X1+G)-
   W*4Ø,64
36 POKE (56225-X1)+(Z*X1+G)-
   W*4Ø,7
37 NEXT G:GOTO 43
38 FOR G=1 TO X1:CA=227:CC=CO:
```





Dragon and Tandy: a plot of percentages

Commodore 64: each bar has a ‘3D’ effect

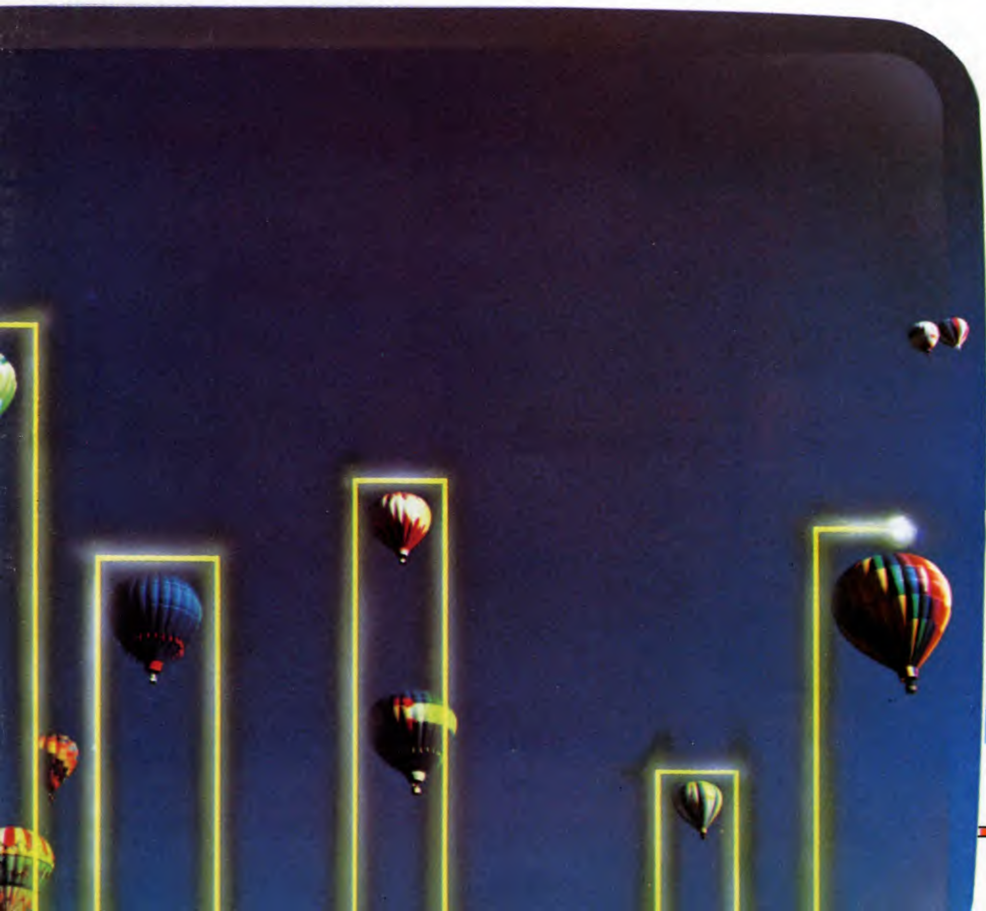
```
   IF G< >1 AND G=X1 THEN CO=1
39 IF ZZ<Ø THEN CA=228
40 IF B(Z)/D>ZZ OR A(Z)/D<ZZ THEN 43
41 POKE((1953-X1)+Z*X1+G)-
   W*4Ø,CA:POKE((56225-X1)+Z*X1+
   G)-W*4Ø,CO
42 CO=CC:NEXT G
43 POKE BB,RND(1)*2+5
44 NEXT ZZ,Z:POKE BB,5
45 PRINT “☰▲▼”;LEFT$(Q$,
   2Ø-ABS(B/D))
46 FOR Z=Ø TO A STEP D*2: IF C=<1Ø
   AND INT(Z)< >Z THEN PRINT“◙◙
   ◙”: GOTO 48
47 PRINT INT(Z)“◙◙◙”
48 NEXT Z
49 PRINT “☰▼”;LEFT$(Q$,2Ø-
   ABS(B/D))
5Ø FOR Z=Ø TO B STEP -D*2: IF C=<1Ø
   AND INT(Z)< >Z THEN PRINT
   “▼”:GOTO 52
51 PRINT INT(Z)“▼”
52 NEXT Z
53 POKE 198,Ø
54 GET A$:IF A$=“” THEN 54
55 GOTO 2ØØ
1ØØ POKE BB,1:POKE BB+1,1
1Ø3 IF FF=Ø THEN 5ØØ
1Ø5 PRINT “♥▒◪”TAB(1Ø)“BAR□
   GRAPH□PLOTTER□”
11Ø PRINT “▼▼□□□▲↑”
115 FOR Z=1 TO 15
12Ø PRINT “□□□▐”:NEXT Z
125 PRINT “□□□⊞○◣”;
13Ø FOR Z=1 TO 3Ø
135 PRINT “-”;:NEXT Z
14Ø PRINT “>”
145 PRINT “☰⊞▼▼▼▼▼▼▼▶
   ▶▶▶▶VALUE:□USERS□UNITS”
15Ø PRINT “◀▼▼▼▼▼▼▼▼▼
   ▼▶▶▶▶▶▶BARS:□WEEKS/
   MONTHS/YEARS”
155 PRINT “▼▼▼▒◪”TAB(9)
   “PRESS□KEY□TO□CONTINUE”
16Ø POKE 198,Ø
165 WAIT 198,1
17Ø GOTO 24
2ØØ POKE BB,6:POKE BB+1,6
21Ø PRINT “♥▼▼▼▼▼▼▼”TAB
   (15)“◣◪□□OPTION□□▼▤”
22Ø PRINT TAB(12)“1□ENTER□DATA”
23Ø PRINT TAB(12)“2□SCALED□GRAPH”
24Ø PRINT TAB(12)“3□VIEW/EDIT□ DATA”
245 PRINT TAB(12)“4□FULL□SCREEN□
   GRAPH”
25Ø PRINT TAB(13)“▼◣◪ENTER□
   CHOICE□?”
26Ø POKE 198,Ø
27Ø GET A$
28Ø ON VAL(A$) GOTO4ØØ,7ØØ,3ØØ, 6ØØ
29Ø GOTO 27Ø
3ØØ IF FF=Ø THEN 5ØØ
3Ø5 POKE BB,7:POKE BB+1,7
31Ø PRINT “♥▒”N$“□DATA:”
315 PRINT,“▼⊞BAR”,“◀HEIGHT▼”
32Ø FOR Z=1 TO XX
325 IF A(Z)< >Ø THEN PRINT,“◀”Z,
   “⊞”A(Z):GOTO 335
33Ø PRINT,“⊞”Z,“◀”B(Z)
335 POKE 198,Ø
34Ø GET A$
345 IF A$=“” THEN 34Ø
35Ø IF A$< >CHR$(13) THEN 37Ø
355 PRINT “◙■”TAB(18);
36Ø INPUT C(Z)
365 GOTO 13
37Ø NEXT Z
375 GOTO 2ØØ
4ØØ POKE BB,13:POKE BB+1,13
41Ø PRINT“♥▼▼▼▼▼▼▼▼
   ▶▶▶▶▶▶▶▲PRESS□(◀E▲)
   NTER□DATA□,□(◀M▲)ENU”
42Ø GET A$
43Ø IF A$=“E” THEN 1
44Ø IF A$=“M” THEN 2ØØ
45Ø GOTO 42Ø
5ØØ PRINT“♥▤”TAB(16)“NO□DATA!”
51Ø FOR Z=1 TO 5ØØ:NEXT Z
52Ø GOTO 2ØØ
6ØØ IF FF=Ø THEN 5ØØ
6Ø5 IF C<=1Ø THEN D=.5:GOTO 1ØØ
6Ø6 IF C<=2Ø THEN D=1:GOTO 1ØØ
61Ø D=C/2Ø:GOTO 1ØØ
7ØØ IF FF=Ø THEN 5ØØ
7Ø5 IF C<=1Ø THEN D=1.25:GOTO 1ØØ
7Ø6 IF C<=2Ø THEN D=2.5:GOTO 1ØØ
71Ø D=C/2Ø+9:D=D*.1:D=INT(D)*
   1Ø:GOTO 1ØØ
```

```
1Ø A=Ø
2Ø MODE1
3Ø VDU19,Ø,4,Ø,Ø,Ø,19,1,2,Ø,Ø,Ø,19,
   2,1,Ø,Ø,Ø
4Ø CLS:PRINTTAB(1Ø,8)“OPTIONS:”
   TAB(1Ø,11)“1□ENTER□DATA”
   TAB(1Ø,13)“2□EDIT□OR□VIEW
   □DATA”TAB(1Ø,15)“3□VIEW
   □GRAPH”TAB(1Ø,17)“4□END”
5Ø G=GET
6Ø IF G=49 THEN PROCINITVAL: GOTO4Ø
7Ø IF G=52 THEN 13Ø
8Ø IF A=1 THEN 1ØØ
9Ø PRINT'“YOU□HAVEN'T□ENTERED
   □ANY□DATA.”:FOR T=1 TO 2ØØØ:
   NEXT:GOTO 4Ø
1ØØ IF G=5Ø THEN PROCEDIT: GOTO 4Ø
11Ø IF G=51 THEN PROCVIEW
12Ø GOTO 4Ø
13Ø CLS
```

```
140 END
150 DEF PROCINITVAL
160 IF A=1 THEN PRINT'"YOU□HAVE
   □ENTERED□SOME□DATA□
   ALREADY":FOR T=1 TO 2000: NEXT:
   ENDPROC
170 A=1
180 CLS
190 INPUT'"TITLE□OF□GRAPH□",
   T$:T$=LEFT$(T$,25)
200 INPUT'"NAME□OF□X-AXIS□",
   X$:X$=LEFT$(X$,25)
210 INPUT'"NAME□OF□Y-AXIS□",
   Y$:Y$=LEFT$(Y$,25)
220 INPUT'"HOW□MANY□BARS□",N
230 IF N=0 THEN 220
240 CLS
250 PRINT'"NOW□ENTER□YOUR□
   DATA"'
260 PRINTTAB(12)"BAR","VALUE"'
270 DX=1000/N
280 DIM Y(N)
290 FOR T=1 TO N
300 PRINTTAB(12)"No.□";T"□□□";:
   INPUT""Y(T)
310 NEXT
320 ENDPROC
330 DEF PROCVIEW
340 MAX=0:MIN=0
350 FOR T=1 TO N
360 IF Y(T)>MAX THEN MAX=Y(T)
370 IF Y(T)<MIN THEN MIN=Y(T)
380 NEXT
390 PROCSCALEPOS
400 PROCSCALENEG
410 PROCNORM
420 CLS
430 GCOL0,3
440 R=MAX3-MIN3
450 MOVE242,60:DRAW242,960
460 DY=900/R
470 YAX=60-MIN3/R*900
480 MOVE242,YAX:DRAW1200,YAX
490 W=1000/N-8
500 PROCYAXIS
510 PROCNAMES
520 FOR T=1 TO N
530 PROCBLOCK
540 NEXT
550 G=GET:ENDPROC
560 DEF PROCBLOCK
570 IF Y(T)>0 THEN GCOL0,1 ELSE
   GCOL0,2
580 MOVE250+(T-1)*DX,YAX:
   MOVE250+W+(T-1)*DX,YAX
590 PLOT85,250+(T-1)*DX,YAX+
   Y(T)*DY/FAC:PLOT85,W+(T-1)
   *DX+250,YAX+Y(T)*DY/FAC
600 ENDPROC
610 DEF PROCSCALEPOS
620 FAC=0
630 IF MAX=0 THEN 650
640 FAC=10∧INT(LOG(MAX))
650 ENDPROC
660 DEF PROCSCALENEG
670 FAC2=0
680 IF MIN=0 THEN 700
690 FAC2= 10∧INT(LOG(-MIN))
700 ENDPROC
710 DEF PROCNORM
720 IF FAC2>FAC THEN FAC=FAC2
730 MAX3=INT(MAX/FAC+1):
   MIN3=INT(MIN/FAC-1)
740 IF MIN=0 THEN MIN3=0
750 IF MAX=0 THEN MAX3=0
760 ENDPROC
770 DEF PROCYAXIS
780 GCOL0,3
790 VDU5
800 FOR T=MIN3 TO MAX3
810 MOVE0,70+(T-MIN3)*DY:
   PRINTTAB(7-LEN(STR$(T)));T
820 NEXT
830 VDU4
840 ENDPROC
850 DEF PROCNAMES
860 GCOL0,3
870 VDU5
880 FOR T=1 TO LEN(Y$)
890 MOVE64,900-T*32:PRINT
   MID$(Y$,T,1)
900 NEXT
910 MOVE320,31:PRINTX$
920 VDU4
930 VDU30:PRINT"X ";FAC;
   TAB(15)T$
940 ENDPROC
950 DEF PROCEDIT
960 VDU12:PRINT"EDITING□OPTION.
   □PRESS□SPACE□BAR□WHEN□
   YOUGET□TO□THE□ONE□YOU□
   WANT□TO□CHANGE□□□□□□□
   TO□MOVE□ON□TO□THE□NEXT□
   ONE□PRESS□RETURN"
970 PRINT'
980 PRINT"□□□□BAR□NO
   □□□□□□□VALUE"
990 FOR T=1 TO N
1000 PRINTT,Y(T)
1010 IF GET<>32 THEN 1030
1020 PRINTT;"□",;CHR$(8);:
   INPUT""Y(T)
1030 NEXT
1040 ENDPROC
```

```
10 PMODE 4,1
20 CLS
30 PRINT@45,"menu":PRINT@102,
   "1-□ENTER□DATA":PRINT@166,
   "2-□DISPLAY□BAR□CHART":PRINT
   @230,"3-□VIEW/EDIT□DATA":
   PRINT@294,"4- QUIT PROGRAM"
40 A$=INKEY$:IF A$<"1" OR
   A$>"4" THEN 40
50 IF A$="1" AND DA=1 THEN 80
60 ON VAL(A$) GOSUB 1000,2000,
   3000,4000
70 GOTO 20
80 PRINT@484,"ARE□YOU□SURE□?";
90 A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 90
100 IF A$="Y" THEN CLEAR200:
   A$="1":GOTO 60
110 GOTO 20
1000 DA=1:CLS:INPUT"□NAME□
   OF□X-AXIS";X$
1010 X$=LEFT$(X$,32):IF X$=
   "" THEN X$="X-AXIS"
1020 MD=INT(16-LEN(X$)/2)
1030 INPUT"□NAME□OF□Y-
   AXIS";Y$
1040 Y$=LEFT$(Y$,12):IF Y$
   ="" THEN Y$="Y-AXIS"
1050 HT=INT(6-LEN(Y$)/2)
1060 INPUT"□NO□OF□BARS□";NB
1070 NB=INT(NB):IF NB<1 THEN
   1060
1080 BL=INT(26/NB):IF BL<1 THEN
   BL=1
1090 DIM A(NB)
1100 PRINT
1110 FOR K=1 TO NB
1120 PRINT"□VALUE□OF□BAR";
   K;:INPUT A(K)
```

```
1130 NEXT
1140 RETURN
2000 IF DA=Ø THEN PRINT@455,
   "no□data□entered":FOR K=1
   TO 2000:NEXT:RETURN
2010 TP=Ø:BT=Ø
2020 FOR K=1 TO NB
2030 IF A(K)>TP THEN TP=A(K)
2040 IF A(K)<BT THEN BT=A(K)
2050 NEXT
2060 IF TP=Ø AND BT=Ø THEN PRINT
   "□ALL□VALUES□ZERO":FOR
   K=1 TO 2000:NEXT:RETURN
2070 CLS:PRINT@64,"DO□YOU□
   WANT□A□SCALED□GRAPH□
   Y/N ?"
2080 A$=INKEY$:IF A$<>"Y"
   AND A$<>"N" THEN 2080
2090 IF A$="N" THEN 2120
2100 IF TP>Ø THEN E=INT(LOG(TP)/
   LOG(1Ø)):TP=INT (1+TP/(1Ø↑E))
   *1Ø↑E
2110 IF BT<Ø THEN E=INT(LOG
   (-BT)/LOG(1Ø)):BT=-
   INT(1-BT/(1Ø↑E))*1Ø↑E
2120 IN=178/(TP-BT)
2130 ST=INT(IN*TP)
2140 T$="":IF TP=Ø THEN 2160
2150 IF TP>ABS(BT) THEN E=2*INT
   (LOG(TP)/LOG(1ØØØ)) ELSE E=
   2*INT(LOG(ABS(BT))/LOG(1ØØØ))
2160 T$="+"+MID$(STR$(INT(.5
   +TP/1Ø↑E)),2):B$="":IF
   BT=Ø THEN 2180
2170 B$=STR$(INT(.5+BT/1Ø↑E))
2180 SL=1:LP=NB:IF NB>26 THEN
   LP=26
2190 POKE 179,243:PCLS:POKE 65475,Ø:
   POKE 65477,Ø:POKE 65479,Ø
2200 POKE 179,2
2210 LINE(4Ø,Ø)-(47,191),PRESET, BF
2220 IF T$="" THEN 2260
2230 FOR K=1 TO LEN(T$):P=ASC
   (MID$(T$,K,1))AND63:FOR M=Ø
   TO 11
2240 POKE 154Ø-LEN(T$)+32*M+K,P
2250 NEXT M,K
2260 IF B$="" THEN 2300
2270 FOR K=1 TO LEN(B$):P=ASC
   (MID$(B$,K,1))AND63:FOR M=Ø
   TO 11
2280 POKE 6916-LEN(B$)+M*32+K,P
2290 NEXT M,K
2300 FOR K=1 TO LEN(X$)
2310 P=ASC(MID$(X$,K,1))AND63
2320 FOR M=182 TO 19Ø:POKE 15Ø3+
   MD+32*M+K,P:NEXT
2330 NEXT
2340 FOR K=1 TO LEN(Y$)
2350 FOR M=Ø TO 11
2360 P=ASC(MID$(Y$,K,1))AND63
2370 POKE 1568+384*(K+HT)+32*M,P
2380 NEXT M,K
2390 FOR K=5 TO 31
2400 POKE 1536+32*ST+K,128
2410 NEXT
2420 FOR K=ST TO Ø STEP -22
2430 POKE 1541+K*32,2Ø2
2440 NEXT
2450 FOR K=ST TO 178 STEP 22
2460 POKE 1541+K*32,2Ø2
2470 NEXT
2480 FOR K=SL TO LP:CO=(CO+1)
   AND1
2490 POKE 178,35+CO*64:POKE 179,
   3+CO*192
2500 IF INT(A(K)*IN)>Ø THEN LINE
   (48+8*(K-SL)*BL,ST-1)
   -(47+8*(K-SL+1)*BL,ST-
   INT(A(K)*IN)),PSET,BF
2510 IF FIX(A(K)*IN)<Ø THEN LINE
   (48+8*(K-SL)*BL,ST)-
   (47+8*(K-SL+1)*BL,ST-FIX
   (A(K)*IN)),PRESET,BF
2520 NEXT K
2530 IF LP=NB THEN 2570
2540 SL=LP+1:LP=LP+26:IF
   LP>NB THEN LP=NB
2550 IF INKEY$="" THEN 2550
2560 POKE 179,243:LINE(48,Ø)
   -(255,178),PRESET,BF:GOTO 2390
2570 IF INKEY$="" THEN 2570
2580 RETURN
3000 IF DA=Ø THEN PRINT@455,
   "no□data□entered":FOR K=1
   TO 2000:NEXT:RETURN
3010 SL=1:LP=NB:IF NB>14 THEN
   LP=14
3020 CLS:PRINT"□bar","value"
3030 FOR K=SL TO LP
3040 PRINTK,A(K)
3050 NEXT
3060 IF NB=LP THEN 3130
3070 PRINT@48Ø,"e□TO□EDIT,□
   ANY□OTHER□TO□CONTINUE";
3080 A$=INKEY$:IF A$="" THEN 3080
3090 IF A$="E" THEN 3170
3100 SL=LP+1:LP=LP+14
3110 IF LP>NB THEN LP=NB
3120 GOTO 3020
3130 PRINT@48Ø,"e□TO□EDIT,
   ANY□OTHER□TO□RETURN";
3140 A$=INKEY$:IF A$="" THEN 3140
3150 IF A$="E" THEN 3170
3160 RETURN
3170 CLS:PRINT:INPUT"□NUMBER□OF□
   ENTRY□TO□BE□EDITED□?";E
3180 E=INT(E):IF E<1 OR E>NB THEN
   3170
3190 PRINT:PRINT"NEW VALUE OF
   ENTRY";E;:INPUT A(E)
3200 GOTO 3020
4000 CLS:PRINT@37,"ARE□YOU□
   SURE□(Y/N)□?"
4010 A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 4010
4020 IF A$="N" THEN RETURN
```



PLANNING AN ADVENTURE

Transport your friends into a fantasy world of your own creation, and give them a few headaches too! We look at the world and history of adventure games

For those of you who want some relief from arcade-type zapping games, there is an alternative—adventure games. In this kind of game the player is totally immersed in a fantasy world created by the programmer. Exercising his wit, wisdom and knowledge of odd facts and figures, he journeys around a fantasy world trying to complete the quest that the writer has dreamed up.

Over the next issues of *INPUT* you'll see how you can write adventure games of your own—but first, an introduction to adventure games and what adventuring is all about.

THE HISTORY OF ADVENTURES

The idea of writing adventure games originally came from the popularity of non-computer games such as Dungeons and Dragons, and a desire to do something with computers more interesting than mere data processing.

In Dungeons and Dragons, players assume the guises of a range of characters and hack one another to pieces (in imagination) in a world—known as a Dungeon—created by the Dungeon Master. In adventure games the programmer assumes a role similar to that of the Dungeon Master, creating a world of his own. The player, on the other hand, assumes a role similar to that of a character in D&D.

Unlike in D&D, adventure players cannot usually choose the traits of their character, but this depends entirely on the game. In some of the more sophisticated versions, they can actually select their equipment, etc., before they set out on their quest. The emphasis is perhaps a little less bloodthirsty, too—although, of course, this is totally up to the writer.

Q+A

What do I do if I get stuck in an adventure?

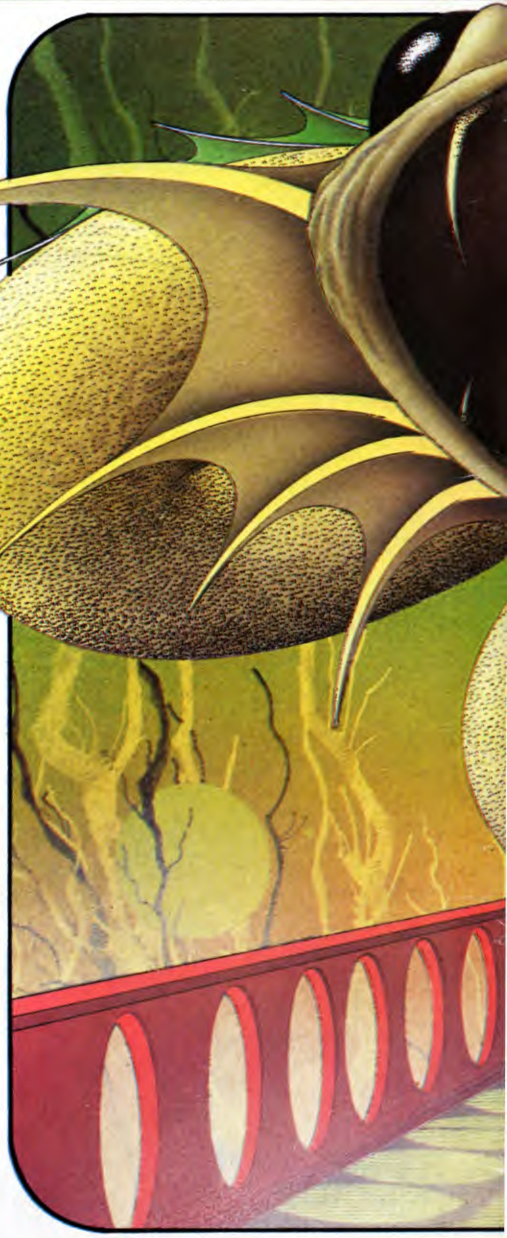
Don't give up, try again after a while as you may come up with a new idea or a new strategy.

If you get really stuck, commercial adventures often provide a hints or answers sheet. It may be supplied in a sealed envelope with the game or you may have to send away for it.

Another source of help is the letters page in some magazines.

The earliest adventure was written on a mainframe in FORTRAN rather than BASIC. The program occupied some 300K of memory —somewhat more than your microcomputer has on board.

The real story begins, though, when Scott Adams transferred some of the ideas to a TRS 80 micro in 1978 and proved that it was quite possible to write a satisfactory adventure in far less memory space. The themes Adams adopted for his games —Adventureland, Pirate's Cove, Mystery Fun House and The Count—have been endlessly reused by adventure writers ever since.



ADVENTURE TYPES

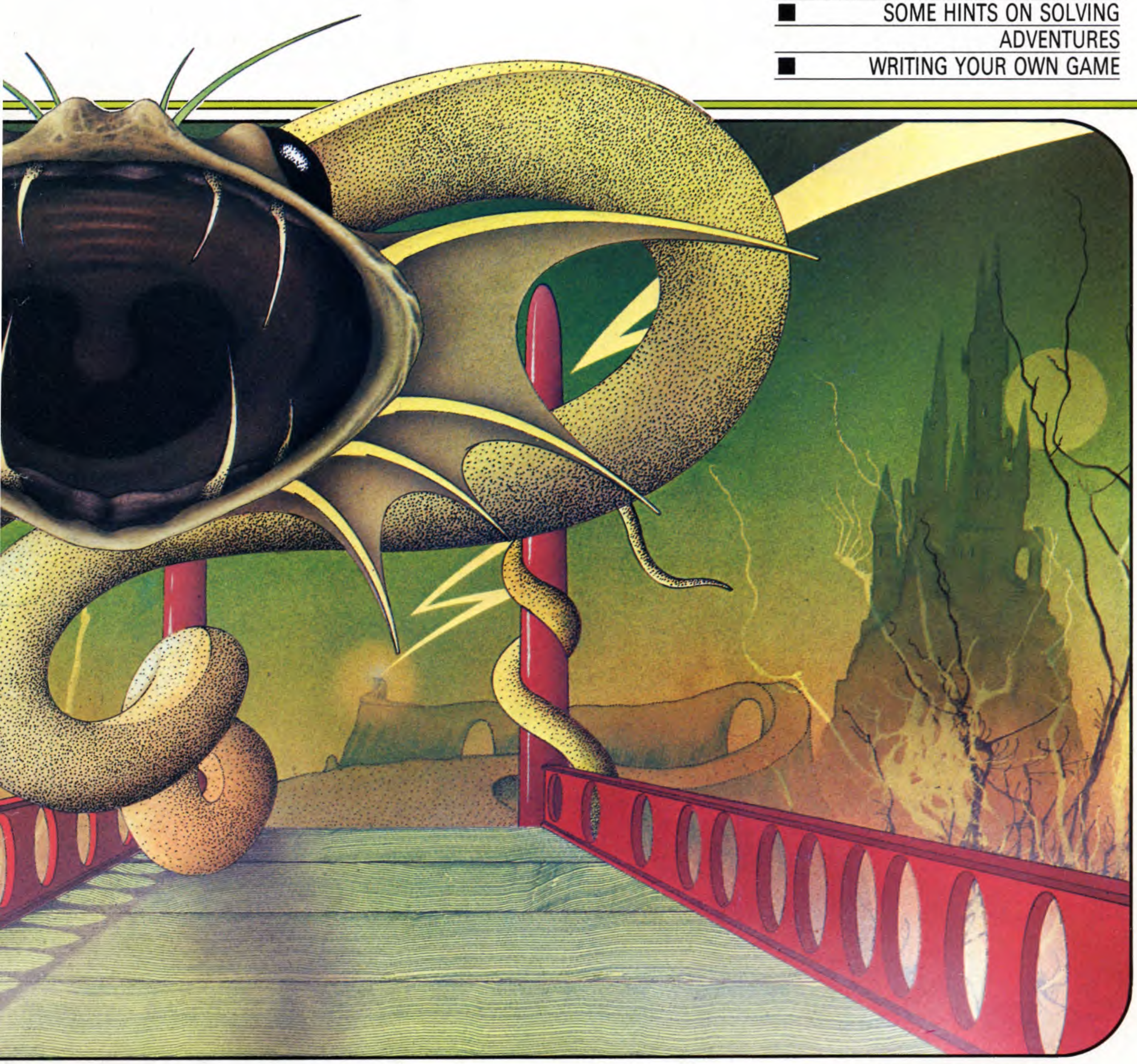
The original mainframe game and Adams' microcomputer games display just text on the screen. These text-only types are still the most popular, and some would say, the best kind of adventure.

Text-only adventures really exist in the players' mind, and as you play a good adventure you'll find that you get totally wrapped-up in the story.

Graphics have to be very sophisticated to compete with your imagination. For example, you can imagine a far fiercer ogre than can be drawn on even the best graphics screen, so it is quite possible that graphics will spoil your enjoyment. A major consideration against the use of graphics is that the display takes up a great deal of memory that could be used for extending the scope of the adventure. It is also all too easy to slow down the adventure because the player will have to wait for the picture to be drawn at each new location.

Certain adventures give you a score for completing certain stages of the quest so that if you get killed at any stage you can judge how well you've done. Some even give you a rating—buffoon or expert, say. The other end of the scale is the adventure in which you are given no clue at all as to how well you are doing, nor how close you are to the final goal. The ultimate satisfaction here comes from solving an endless series of puzzles and gradually coming closer and closer to finding the end of the quest and solving the adventure.

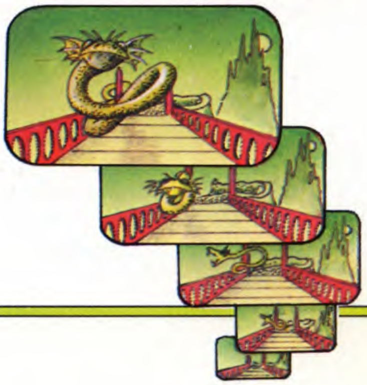
- WHAT IS AN ADVENTURE GAME
- HOW TO PLAY
- SOME HINTS ON SOLVING ADVENTURES
- WRITING YOUR OWN GAME



PLAYING ADVENTURES

When you RUN an adventure you'll normally be told something about the world you find yourself in—it may be some exotic location on Earth, a planet far out in the galaxy, or in some total fantasy land. The game may take place in the past, present or future—or even a mixture of the three. You'll usually be told some helpful facts or background information, such as who rules the world, who you are (if you are playing a character role), something about your friends and enemies, and most importantly, what you must do to solve the adventure, and/or win the game. Read the instructions carefully because there's normally lots of important information.

After that the first location description will appear. It will probably say something like this:



S
W
E
WEST
NORTH
EAST

```
YOU ARE STANDING NEXT TO A VERY LARGE
CAULDRON FULL TO THE BRIM WITH
BUBBLING GREEN FLUID. THERE IS AN EVIL
SMELL IN THE AIR. A SPOON LIES ON THE
GROUND.
YOU CAN GO EAST
           WEST
           NORTH
WHAT NOW?
```

You now have to decide what you want to do. Should you use the spoon to stir the fluid, or even try drinking some? Should you leave it alone? Or should you explore, looking for a bottle or a container for the green fluid so that you can carry some with you?

Should you decide to use the spoon you need to type something like this:

```
GET SPOON
```

to which the computer will reply OK, or YOU CANNOT GET THE SPOON—YET!, or some other message.

At each stage of the game, you must tell the computer exactly what you want to do. *How* you tell it depends on the game. Most games will expect you to type in your instructions to the computer as a verb followed by a noun—e.g. GET SPOON, STRANGLE ELEPHANT, UPROOT TREE and so on.

More sophisticated games will accept whole sentences, but this is the exception rather than the rule. These kind of games allow you to say something like KILL THE STICK INSECT BY STEPPING ON IT WHILE SINGING 'ALL YOU NEED IS LOVE'. A program that will take instructions as complicated as this will have to be very complex in itself, and is well outside the scope of a beginner.

Most adventures will understand—or even expect— shortened versions of words. For example, it is quite normal to type N instead of NORTH in an adventure. Using this type of abbreviation you can speed up the game and save memory space.

Directions may not be just N, S, E, and W. You may have to go UP or DOWN, or even NE, SE, SW and NW. If the game doesn't tell you which directions are open to you, don't forget to try all possibilities.

The normal pattern is that the world of the adventure is based on a grid of possible locations, usually in a square. What these locations represent is at the whim of the programmer—they may be rooms in a castle, or underground chambers in a mine. The link between the locations may be something obvious, like a door or a path or a flight of stairs, or it may be more obscure—a river you have to swim, for example.

SOLVING ADVENTURES

There is normally only one solution to an adventure—collecting all the treasure and taking it to the Golden Gate, or slaughtering the Iron Maiden and escaping unscathed—and a fixed sequence of problems to solve. The odds are that you'll need many, many attempts before the adventure is finished. In fact, any adventure which doesn't take you days, weeks or even months of sweat isn't much good!

There are some basic rules and hints that will help you to solve most adventures a little quicker.

Almost without exception the objects you will find in adventures will have some use. It's a waste of memory on the part of the programmer to plant too many complete red herrings—although beware that some objects may be 'double-edged swords'. For example, you might need to be carrying a bag of gold coins in order to pass over a toll bridge, but if you decided to swim the river, their weight would make you sink. In general, pick up as many objects as you can, but sometimes you'll find that you can carry only a certain number of objects.

Most objects are usually needed only once in an adventure. An exception would be something like a sword which can be used many times for bumping off baddies. If you are limited in the number of objects that you can carry, bear in mind that it's usually safe to discard objects once they've been used.

Always draw a map. Mark on it the room names, anything of interest about the room, any objects that are in the room, *all* the entrances and exits, and their directions.

The map will save you a lot of time and effort when you are retracing your steps—something you'll have to do many times during a game. If you have to leave any objects behind because you can't carry everything, don't forget to mark their position on the map. Just as importantly, drawing a map will enable you to be sure of exploring the whole adventure—and, for that matter, will stop you sinking into the quicksand for the umpteenth time.

Most games allow you to call for an inventory of the objects you're carrying. When confronted with a puzzle it's a good idea to see exactly what objects you have to hand by typing INVENTORY, INVE or simply I, depending on the adventure.

Similarly, some adventures allow you to ask for help—the formula depends again on the game. You may or may not get a helpful hint—more often than not you'll get something like NO HELP HERE.

Some games follow the plot of a particular book very closely, and in that case studying the book in question is definitely a good idea. Other games have small sections or ideas which have been borrowed. If you think you recognise something, and cannot solve a particular problem, try searching out the book. Similarly, if a mean-looking character with a huge chopper blocks your way and asks for the diameter of the Earth, don't guess, go and look the thing up!

A further device much-used in adventure games is a selection of puns. Look out for them, everything may not be quite as it seems.

It may also be a good idea to keep a Thesaurus—or similar directory of synonyms—at hand so that you can try as many variations on a particular phrase as possible. The programmer might not have included POLISH as well as RUB, for example.

And one last tip. If the adventure you are playing allows you to SAVE it part-way through, and you are about to attempt something hazardous, then SAVE it before you try. If you do get killed, then you can simply continue from where you left off. It'll also allow you many attempts at, say, slaying a dragon, crossing a rickety bridge or escaping from a cavern.

WRITING ADVENTURES

Writing adventures is a very good way of getting to grips with BASIC. Nearly all the important aspects of the language are utilised—string handling and slicing, the various forms of PRINT for screen formatting, variables, arrays and so on.

Most commercial adventures are still written in BASIC because there's no real need for the speed of machine code. The real barrier against you producing absolutely top-quality games is your own imagination.

Before you sit down to program your adventure, though, you will need to have a very clear idea of what you are going to do and what your adventure is about. The plot, puzzles, perils, and so on must all be preplanned if you are to save yourself a lot of trouble.

First, get yourself some paper and try noting down some ideas. Don't worry if you haven't yet got a full vision of what an adventure game entails—what you need is an idea for a story, a location for the adventure, and some dastardly puzzles for the player to solve. As this part of Games Programming progresses you'll see how an idea for a game can be turned into an adventure program and learn how to adapt your own, original ideas.

Be very careful with your chosen world. Try to make it as interesting as possible—you'd have to work very hard at making an adventure set in an office block at all enjoyable.

You can look towards books, films, TV and plays or even news stories for inspiration. You can also borrow some ideas from other adventures, but probably the best source of inspiration is a slightly warped mind! Always aim for a theme or central idea, though, that you can develop throughout the adventure.

Try to get the right balance between challenge and impossibility. It's no good spending a lot of time and effort writing an adventure that everyone can solve within half an hour. Conversely, you won't make many friends if your adventure is totally impossible to solve. The basic rule is 'give the punters a chance'—but not too much!!

Try not to leave too many empty rooms—locations—in your adventures. They do not really add anything to the adventure, and occupy vital memory space. Empty rooms also make for a boring adventure.

Don't make your early adventures too complex because the problems of adventures can prove very difficult to debug until you have had some practice. Get to know what's involved before you attempt anything very ambitious. Keep an eye on how much memory there is remaining on your machine.

In the adventure that you'll see building up in the later parts of *INPUT* there are lots of REM statements. In order to conserve memory in a large adventure, it's best to do without all of these, although they make it easier to write the adventure in the first place.

A further place for memory economies can be in the room descriptions. Don't make them too short, though, because you may lose all the flavour of the adventure. It's really up to you to strike the correct balance between flavour and space.

Next week you'll see how to turn an idea for an adventure into a map and start on the adventure program.

Q+A

How much memory do I need to write an adventure game?

The short answer is 'as much as possible'.

The adventure which builds up over the next issues of *INPUT* is very simple and occupies roughly 5K of BASIC; and you'll soon see that it's not of the best commercial standard.

The best adventures have a very large number of rooms and puzzles and are difficult because of the cumulative effect of the problems. With a limited amount of memory you will be forced to set a smaller number of very much more difficult problems if the adventurer is not to solve the game immediately, and this is not really very satisfactory.

A computer with 16K of memory would be the absolute minimum.

HOW MUCH MEMORY REMAINS?

When you are writing a large adventure game, it is easy to find that you are pushing the limits of your machine's memory. Obviously, the problems of running out of memory are more acute in the case of machines with smaller memories—in fact, it could be a waste of time trying to write an adventure for a computer with less than 16K.

For each of the machines there is a way of checking how much memory remains with just a simple routine.

On the Spectrum, type:

```
PRINT (PEEK 2373Ø+256*PEEK 23731)
   −(PEEK 23653+256*PEEK 23654)
```

This takes into account both the program itself and the variable storage. The best count of how much memory remains, therefore will be after the program has been RUN—if this is possible during program development.

On the Commodores, the remaining memory is obtained by typing:

```
PRINT FRE(Ø)
```

The remaining memory takes into account both the space occupied by the program and the space occupied by variables. The best way of finding out the exact amount of memory remaining is to RUN the program—if this is possible during development.

On the Acorn machines, the remaining memory is obtained by typing:

```
PRINT HIMEM−TOP
```

This number only includes the space occupied by the program—variable space is not included so you will have to allow a little extra when estimating the size of your program. For the greatest amount of memory use Mode 7 on the BBC and Mode 6 on the Electron.

On the Dragon and Tandy, the remaining memory can be found by typing:

```
PRINT MEM
```

This takes into account the space occupied by the program, and space occupied by variables. So it's best to RUN the program during development, if that is possible, to get a true indication of the total memory used by the program.

There is a very easy way of increasing the amount of memory available in the Dragon for your BASIC programs. Before you start programming, type in these two simple POKEs:

```
POKE 25,6
POKE 26,1
NEW
```

This gains you some 6K extra for your program because it fools the machine into putting the BASIC program into space normally reserved for high resolution graphics—four 1½K pages at switch-on.

If you do use the POKEs, make sure that there are no graphics commands in your adventure. If there are, the program will become corrupted.

When you SAVE the program, and want to LOAD it some time later, don't forget to POKE and NEW first.

CROSS-REFERENCING YOUR ARRAYS

- SETTING UP AN ARRAY
- ENTERING THE DATA
- WRITING THE PROGRAMS
- EXTRACTING INFORMATION
- ADAPTING THE ARRAY

If you've a lot of interrelated data the information can often be lost amongst all the figures. But put it into an array and the computer can pick out exactly what you want

The article on page 152 showed how you can use an array to hold information within a program—a very useful tool indeed. But a *two-dimensional* array can often prove to be still more useful, because it is an extremely effective way of storing a lot of *interrelated* data.

The best model of a two-dimensional array is to think of it as a stack of pigeon holes on a wall, with each pigeonhole holding a single piece of data. This is a convenient way of storing data as you can easily refer to a particular element simply by pointing to its row and column.

The array can store actual items—such as the screws below—where you can immediately point to the box containing 38mm screws made of brass, or whatever else you are looking for. In this case the data in the array is the number of screws in each box. The array can also hold abstract information such as the number of car dealers for each make of car in each country. Abstract information like this is often collected as the result of a survey and that is the basis of the program given in this article. However, the structure of the program is the same for any two-dimensional array—no matter what type of information it contains.

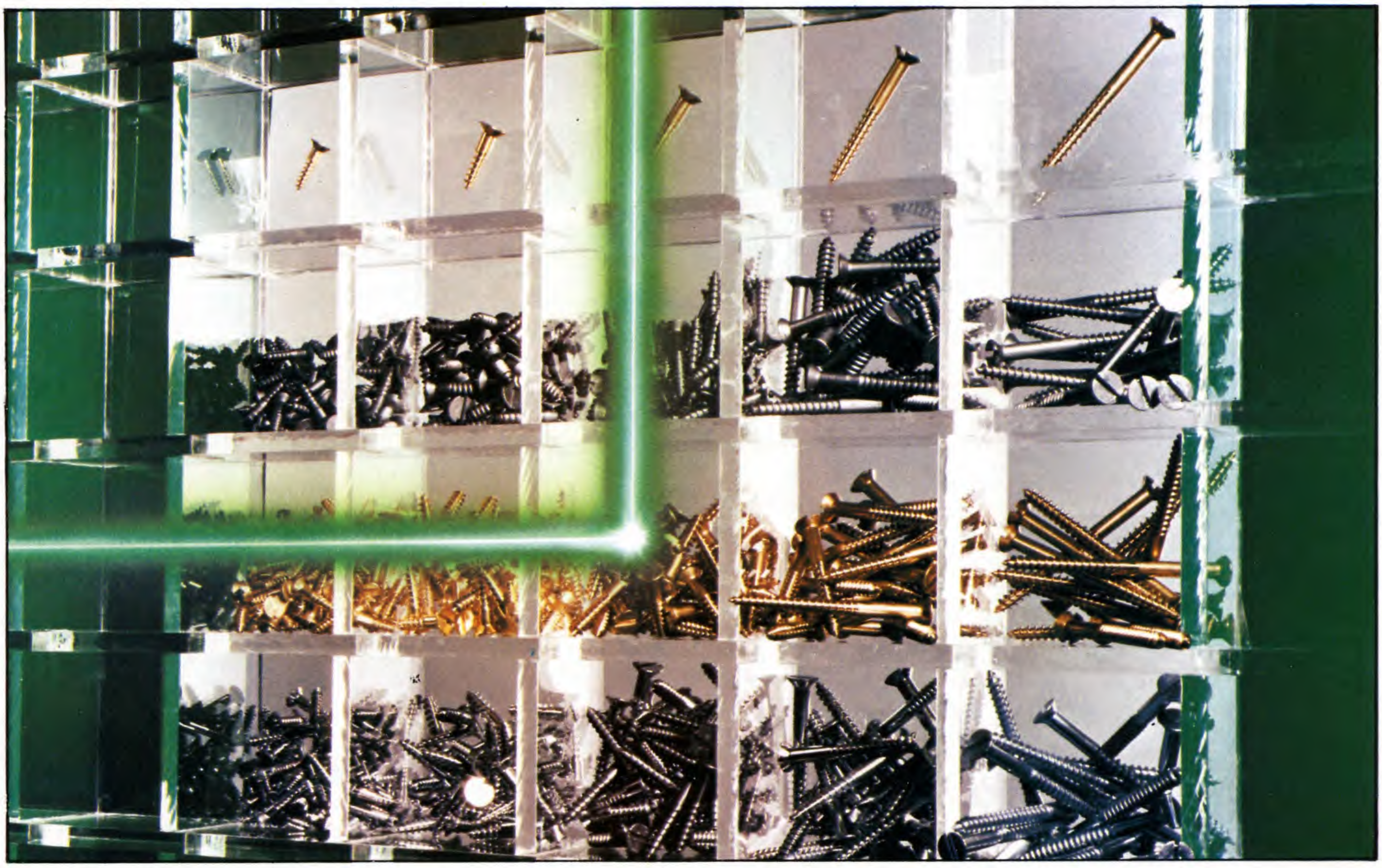
As an example, say you were a teacher doing a survey of the pets kept by a group of children. You'd first write out a list of the children and then write down how many pets of each type they had. The list might look something like this:

Sally: 2 budgies, 1 rabbit
Jason: 1 dog, 4 goldfish
Kevin: 2 hamsters, 1 cat
Josie: 1 dog, 1 cat, 1 hamster
Bert: 1 gerbil, 1 rabbit, 2 goldfish

However, in a list like this it is rather difficult to extract information like "how many people keep cats" or "who has more than four pets"—and the bigger the list gets, the harder this becomes.

A much better way would be to write out the information in the form of a table or chart with the names of the animals along the top and the children down the side. You may not have realised it but what you have just done is set up a two-dimensional array!

1. Arrays are handy for sorting small items like these screws



In a small example like this it is fairly easy to work out the answers to any questions in your head. Where the computer comes in handy, though, is when you have lots of data. If you surveyed a whole class or even a whole school, for example, it would take you rather a long time to work out who had at least one cat or the names of all those who kept hamsters. But the computer could do it in seconds.

SETTING UP THE ARRAY

The next step is to put the information into the computer. If you look at pages 152–155 you'll see how to set up and use a simple one-dimensional array. Two-dimensional arrays are very similar and only involve a little more work to set them up.

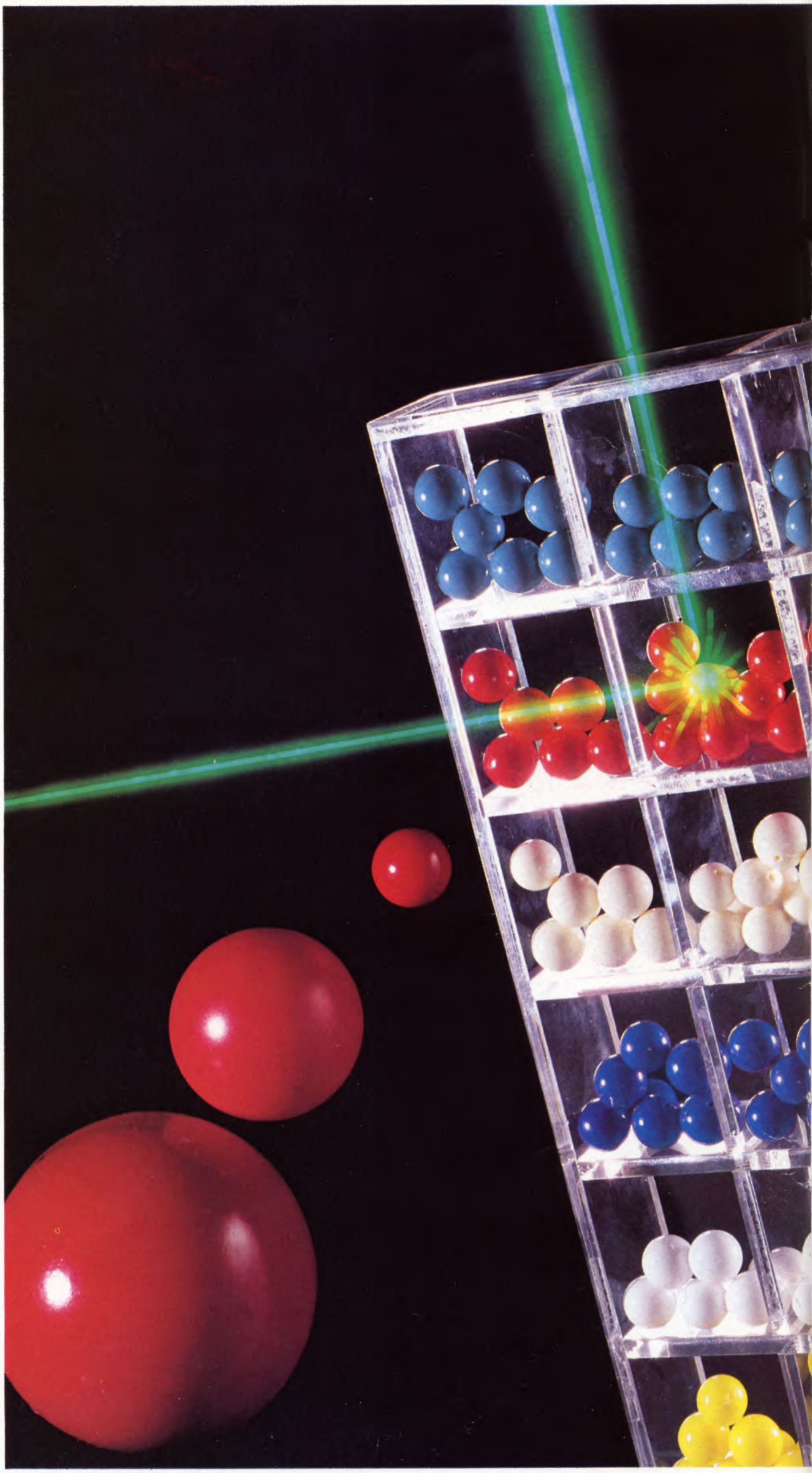
A simple array can be written in the form A(N), where N is the number of elements in the array. A two-dimensional array is written as A(R,C), where R is the number of rows and C is the number of columns. Actually, it doesn't matter if you write it as A(C,R) with the columns before the rows. But you must be consistent and stick to either one or the other.

Now, even with a two-dimensional array, you'll usually want to set up two simple arrays to hold the heading for the columns and the rows—in our case one array holds the names of the children and another holds the names of the pets. The two-dimensional array itself is used to hold the data.

Lets call the first two arrays PET$(C) and CH$(R) and the data array N(R,C). The other array, PT(R), holds the total number of pets in each row. On the Spectrum the arrays are labelled p$(c), c$(r), n(r,c) and p(r) since only single-letter names are allowed. The program will not work on the ZX81 because of the difficulty of entering all the data.

Here's the program to DIMension the arrays and READ in the DATA:

```
100 C=7: R=5
110 DIM PET$(C), CH$(R), N(R,C),
    PT(R)
120 FOR J=1 TO C: READ PET$(J): NEXT
130 FOR J=1 TO R: READ CH$(J): NEXT
140 FOR J=1 TO R
150 FOR K=1 TO C
160 READ N(J,K): NEXT K: NEXT J
3000 DATA BUDGIE,CAT,DOG,FISH,
    GERBIL,HAMSTER,RABBIT
3010 DATA SALLY,JASON,KEVIN,
    JOSIE,BERT
3020 DATA2,Ø,Ø,Ø,Ø,Ø,1
3030 DATAØ,Ø,1,4,Ø,Ø,Ø
3040 DATAØ,1,Ø,Ø,Ø,2,Ø
3050 DATAØ,1,1,Ø,Ø,1,Ø
3060 DATAØ,Ø,Ø,2,1,Ø,1
```



2. The beads are most conveniently arranged in rows of the same colour and also in columns containing the same size

```
100 LET c=7: LET r=5
110 DIM p$(c,7): DIM c$(r,5): DIM
    n(r,c): DIM p(r)
120 FOR j=1 TO c: READ p$(j): NEXT j
130 FOR j=1 TO r: READ c$(j): NEXT j
140 FOR j=1 TO r
150 FOR k=1 TO c
160 READ n(j,k): NEXT k: NEXT j
3000 DATA "BUDGIE", "CAT", "DOG",
    "FISH","GERBIL","HAMSTER",
    "RABBIT"
3010 DATA "SALLY","JASON","KEVIN",
    "JOSIE","BERT"
3020 DATA 2,0,0,0,0,0,1
3030 DATA 0,0,1,4,0,0,0
3040 DATA 0,1,0,0,0,2,0
3050 DATA 0,1,1,0,0,1,0
3060 DATA 0,0,0,2,1,0,1
```

The variables R and C were used to make it easier for you to adapt the program—after all, it's unlikely that your own survey will have the same number of rows and columns as this one. This way all you have to do is change the numbers in Line 100, and the information in the DATA lines of course.

Notice how the data is entered. There's one line for the headings to the columns, another for the headings to the rows then one line for each row of the central array. Also note that you must enter one item of data for each space in the array even if it's zero, or the computer will respond with an error message.

Now that the data is inside the computer you must decide what you want to do with it. You may as well give yourself as many options as possible, after all, it is the computer that's going to be doing all the work, not you.

THE MENU

It's best to write out the options in the form of a menu. And for the pets survey you'll probably want the following:

```
300 LET found=0: CLS
310 PRINT '"MENU"
320 PRINT '"1 LIST PETS"
330 PRINT "2 LIST CHILDREN"
340 PRINT "3 ENTER TYPE OF PET"
350 PRINT "4 ENTER NAME OF
    CHILD"
360 PRINT "5 ENTER NUMBER
    OF PETS"
370 PRINT "6 DISPLAY ARRAY"
380 PRINT '"CHOOSE OPTION"
390 INPUT a
395 IF a<1 OR a>6 THEN GOTO 390
400 CLS
410 GOSUB (a*100+400)
420 GOTO 300
```

```
300 FOUND=0:PRINT "♥"
310 PRINT "MENU■"
320 PRINT "1 LIST PETS"
330 PRINT "2 LIST CHILDREN"
340 PRINT "3 ENTER TYPE OF PET"
350 PRINT "4 ENTER NAME OF
    CHILD"
360 PRINT "5 ENTER NUMBER OF PETS"
370 PRINT "6 DISPLAY ARRAY"
380 PRINT "■CHOOSE OPTION"
390 INPUT A
395 IF A<1 OR A>6 THEN GOTO 390
400 PRINT "♥"
410 ON A GOSUB 500,600,700,800,
    900,1000
420 GOTO 300
```

```
300 FOUND=0:CLS
310 PRINT'"MENU"
320 PRINT'"1 LIST PETS"
330 PRINT "2 LIST CHILDREN"
340 PRINT "3 ENTER TYPE OF PET"
350 PRINT "4 ENTER NAME OF
    CHILD"
360 PRINT "5 ENTER NUMBER OF PETS"
370 PRINT "6 DISPLAY ARRAY"
380 PRINT'"CHOOSE OPTION"
390 INPUT A
395 IF A<1 OR A>6 THEN 390
400 CLS
410 ON A GOSUB 500,600,700,800,
    900,1000
420 GOTO 300
```

```
300 FOUND=0:CLS
310 PRINT@45,"MENU"
320 PRINT@100,"1 LIST PETS"
330 PRINT@132,"2 LIST CHILDREN"
340 PRINT@164,"3 ENTER
    TYPE OF PET"
350 PRINT@196,"4 ENTER NAME OF
    CHILD"
360 PRINT@228,"5 ENTER NUMBER
    OF PETS"
370 PRINT@260,"6 DISPLAY ARRAY"
380 PRINT:PRINT"CHOOSE OPTION ";
390 INPUT A
395 IF A<1 OR A>6 THEN 390
400 CLS
410 ON A GOSUB 500,600,700,800,
    900,1000
420 GOTO 300
```

WRITING THE SUBROUTINES

This, at least, will do for a start although you can always add some more options later on.

The first option simply prints out the list of pets by looping through the array PET$() or p$(). And the second option does much the same for the list of children.

```
499 REM **OPTION 1**
500 PRINT ''"LIST OF PETS"
510 PRINT ''
520 FOR j=1 TO c
530 PRINT p$(j)
540 NEXT j
550 PRINT ''"PRESS A KEY TO RETURN
   TO MENU"
560 PAUSE Ø
570 RETURN
599 REM **OPTION 2**
600 PRINT ''"LIST OF CHILDREN"
610 PRINT ''
620 FOR j=1 TO r
630 PRINT c$(j)
640 NEXT j
650 PRINT ''"PRESS A KEY TO RETURN
   TO MENU"
660 PAUSE Ø
670 RETURN
```

```
499 REM ## OPTION 1 ##
500 PRINT "LIST OF PETS"
510 PRINT
520 FOR J=1 TO C
530 PRINT PET$(J)
540 NEXT J
550 PRINT "■PRESS A KEY TO
   RETURN TO MENU"
560 GET A$:IF A$="" THEN 56Ø
570 RETURN
599 REM ## OPTION 2 ##
600 PRINT "LIST OF CHILDREN"
610 PRINT
620 FOR J=1 TO R
630 PRINT CH$(J)
640 NEXT J
650 PRINT "■PRESS A KEY TO
   RETURN TO MENU"
660 GET A$:IF A$="" THEN 66Ø
670 RETURN
```

```
499 REM**OPTION 1**
500 PRINT''"LIST OF PETS"
510 PRINT''
520 FOR J=1 TO C
530 PRINT PET$(J)
540 NEXT
550 PRINT''"PRESS A KEY TO RETURN
   TO MENU"
560 A=GET
570 RETURN
599 REM**OPTION 2**
600 PRINT''"LIST OF CHILDREN"
610 PRINT''
620 FOR J=1 TO R
630 PRINT CH$(J)
640 NEXT
650 PRINT''"PRESS A KEY TO RETURN
   TO MENU"
660 A=GET
670 RETURN
```

```
499 REM **OPTION 1**
500 PRINT@99,"LIST OF PETS"
510 PRINT:PRINT
520 FOR J=1 TO C
530 PRINT PET$(J)
540 NEXT
550 PRINT:PRINT"PRESS A KEY TO
   RETURN TO MENU"
560 IF INKEY$="" THEN 56Ø
570 RETURN
599 REM **OPTION 2**
600 PRINT@99,"LIST OF CHILDREN"
610 PRINT:PRINT
620 FOR J=1 TO R
630 PRINT CH$(J)
640 NEXT
650 PRINT:PRINT"PRESS A KEY TO
   RETURN TO MENU"
660 IF INKEY$="" THEN 66Ø
670 RETURN
```

Option three is more interesting. If you type in the name of a pet, the computer prints out a list of everybody who has at least one of those pets, and also tells you how many they have:

```
699 REM **OPTION 3**
700 PRINT ''"ENTER TYPE OF PET"
705 DIM i$(7): INPUT LINE i$
710 PRINT '"PEOPLE WHO HAVE A□";i$
720 FOR j=1 TO c
730 IF p$(j)=i$ THEN LET found=j
740 NEXT j
750 IF found=Ø THEN PRINT "PET NOT
   FOUND. TRY AGAIN": GOTO 7ØØ
760 FOR j=1 TO r
770 IF n(j,found)>Ø THEN PRINT
   c$(j);"□";n(j, found)
775 NEXT j
780 PRINT ''"PRESS A KEY TO RETURN
   TO MENU"
785 PAUSE Ø
790 RETURN
```

```
699 REM ## OPTION 3 ##
700 PRINT "ENTER TYPE OF PET"
705 PRINT "THIS LISTS EVERYONE
   WHO HAS A"
710 INPUT P$
715 PRINT
720 FOR J=1 TO C
730 IF PET$(J)=P$ THEN FOUND=J
740 NEXT J
750 IF FOUND=Ø THEN PRINT "PET
   NOT FOUND. TRY AGAIN":GOTO 7ØØ
760 FOR J=1 TO R
770 IF N(J,FOUND)>Ø THEN PRINT
   CH$(J);"□";N(J,FOUND)
```

3. In this array you can see at a glance how many car dealers there are in each country for each make

```
775 NEXT J
780 PRINT "■PRESS A KEY TO
   RETURN TO MENU"
785 GET A$:IF A$ = "" THEN 785
790 RETURN
```

```
699 REM**OPTION 3**
700 PRINT'"ENTER TYPE OF PET"
705 PRINT "THIS LISTS EVERYONE
   WHO HAS A"
710 INPUT P$
715 PRINT'
720 FOR J=1 TO C
730 IF PET$(J)=P$ THEN FOUND=J
740 NEXT
750 IF FOUND=Ø PRINT "PET NOT
   FOUND. TRY AGAIN":GOTO 7ØØ
760 FOR J=1 TO R
770 IF N(J,FOUND)>Ø PRINT
   CH$(J);"□";N(J,FOUND)
775 NEXT
780 PRINT'"PRESS A KEY TO RETURN
   TO MENU"
785 A=GET
790 RETURN
```

```
699 REM **OPTION 3**
700 PRINT@99,"ENTER TYPE OF PET"
705 PRINT "THIS LISTS EVERYONE
   WHO HAS A□";
710 INPUT P$
715 PRINT:PRINT
720 FOR J=1 TO C
730 IF PET$(J)=P$ THEN FOUND=J
740 NEXT
750 IF FOUND=Ø THEN PRINT "PET NOT
   FOUND. TRY AGAIN":GOTO 7ØØ
760 FOR J=1 TO R
770 IF N(J,FOUND)>Ø THEN PRINT
   CH$(J);"□";N(J,FOUND)
   775 NEXT
   780 PRINT:PRINT"PRESS ANY KEY TO
      RETURN TO MENU"
      785 IF INKEY$="" THEN 785
         790 RETURN
```

This routine is more complicated, so it's worth explaining in more detail.

Line 71Ø stores your input in the variable P$ (i$ on the Spectrum). Let's say, for example, that you input "CAT", so P$="CAT" (or i$="CAT"). Lines 72Ø to 74Ø then loop through the whole list of pets looking for one called CAT. If it's found the variable FOUND is set to the column number it was found in. In our case FOUND=2 because CAT is in column 2.

By the time the computer gets to Line 75Ø, if FOUND equals zero (it was set to zero by Line 3ØØ) then the animal you requested was not in the list and you are given another go.

Lines 76Ø to 775 now loop through each element in column 2. If any element is greater than zero it means the person in that row has a cat so their name is printed out along with how many cats they have.

Look at the program and see if you can follow what the computer is doing at each step.

Option four allows you to enter the name of a person to see a list of all their pets. It works in almost exactly the same way as the last routine except that this time the computer looks along a particular row of the array, rather than down a column as before.

```
799 REM **OPTION 4**
800 PRINT ″“ENTER NAME OF CHILD”
805 DIM f$(5): INPUT LINE f$
810 PRINT “PETS BELONGING
    TO ”;f$
815 PRINT ″
820 FOR j=1 TO r
830 IF c$(j)=f$ THEN LET found=j
840 NEXT j
850 IF found=Ø THEN PRINT
    ″“PERSON NOT FOUND. TRY AGAIN”:
    GOTO 8ØØ
860 FOR j=1 TO c
870 IF n(found,j)>Ø THEN PRINT
    n(found,j);“□”;p$(j)
875 NEXT j
880 PRINT ″“PRESS A KEY TO RETURN
    TO MENU”
885 PAUSE Ø
890 RETURN
```

```
799 REM ## OPTION 4 ##
800 PRINT “ENTER NAME OF CHILD”
805 PRINT “THIS LISTS ALL PETS
    BELONGING TO”
810 INPUT F$
815 PRINT
820 FOR J=1 TO R
830 IF CH$(J)=F$ THEN FOUND=J
840 NEXT J
850 IF FOUND=Ø THEN PRINT
    “PERSON NOT FOUND. TRY AGAIN”:
    GOTO 8ØØ
860 FOR J=1 TO C
870 IF N(FOUND,J)>Ø THEN
    PRINT;N(FOUND,J);“□”;PET$(J)
875 NEXT J
880 PRINT “PRESS A KEY TO
    RETURN TO MENU”
885 GET A$:IF A$=“” THEN 885
890 RETURN
```

```
799 REM**OPTION 4**
800 PRINT″“ENTER NAME OF CHILD”
805 PRINT “THIS LISTS ALL PETS
    BELONGING TO”
810 INPUT F$
815 PRINT″
820 FOR J=1 TO R
830 IF CH$(J)=F$ THEN FOUND=J
840 NEXT
850 IF FOUND=Ø PRINT″“PERSON
    NOT FOUND. TRY AGAIN”:
    GOTO 8ØØ
860 FOR J=1 TO C
870 IF N(FOUND,J)>Ø PRINT;N
    (FOUND,J);“□”;PET$(J)
875 NEXT
880 PRINT″“PRESS A KEY TO RETURN
    TO MENU”
885 A=GET
890 RETURN
```

```
799 REM **OPTION 4**
800 PRINT@99,“ENTER NAME OF A
    CHILD”
805 PRINT “THIS LISTS ALL PETS
    BELONGING TO ”;
810 INPUT F$
815 PRINT:PRINT
820 FOR J=1 TO R
830 IF CH$(J)=F$ THEN FOUND=J
840 NEXT
850 IF FOUND=Ø THEN PRINT:PRINT
    “PERSON NOT FOUND. TRY AGAIN”:
    GOTO 8ØØ
860 FOR J=1 TO C
870 IF N(FOUND,J)>Ø THEN PRINT
    N(FOUND,J);“□”;PET$(J)
875 NEXT
880 PRINT:PRINT“PRESS A KEY
    TO RETURN TO MENU”
885 IF INKEY$=“” THEN 885
890 RETURN
```

Option 5 asks you to enter a number then it prints out a list of everyone who has at least that many pets. It also tells you how many they have. So type in the next subroutine now:

```
899 REM **OPTION 5**
900 PRINT ″“ENTER A NUMBER TO LIST
    EVERYONE WITH AT LEAST THAT
    MANY PETS”
910 INPUT a
915 PRINT ″
920 FOR j=1 TO r
925 FOR k=1 TO c
930 LET p(j)=p(j)+n(j,k)
935 NEXT k
940 IF p(j)>=a THEN PRINT c$(j);
    “□”;p(j): LET found=1
945 LET p(j)=Ø
950 NEXT j
955 IF found=Ø THEN PRINT
    ″“NO-ONE HAS□”;a;“□OR
    MORE PETS”
960 PRINT ″“PRESS A KEY TO RETURN
    TO MENU”
970 PAUSE Ø
980 RETURN
```

```
899 REM ## OPTION 5 ##
900 PRINT “ENTER A NUMBER TO LIST
    EVERYONE WITH AT LEAST THAT
    MANY PETS”
910 INPUT A
915 PRINT
920 FOR J=1 TO R
925 FOR K=1 TO C
930 PT(J)=PT(J)+N(J,K)
935 NEXT K
940 IF PT(J)>=A THEN PRINT
    CH$(J);“□”;PT$:FOUND=1
945 PT(J)=Ø
950 NEXT J
955 IF FOUND=Ø THEN PRINT “NO-
    ONE HAS”;A;“OR MORE PETS”
960 PRINT “■PRESS A KEY TO
    RETURN TO MENU”
970 GET A$:IF A$=“” THEN 97Ø
980 RETURN
```

```
899 REM**OPTION 5**
900 PRINT″“ENTER A NUMBER TO LIST
    EVERYONE”″“WITH AT LEAST THAT
    MANY PETS”
910 INPUT A
915 PRINT″
920 FOR J=1 TO R
925 FOR K=1 TO C
930 PT(J)=PT(J)+N(J,K)
935 NEXT K
940 IF PT(J)>=A THEN PRINT
    CH$(J);“□”;PT(J): FOUND=1
945 PT(J)=Ø
950 NEXT J
955 IF FOUND=Ø PRINT″
    “NO-ONE HAS□”;A;“□OR
    MORE PETS”
960 PRINT″“PRESS A KEY TO RETURN
    TO MENU”
970 A=GET
980 RETURN
```

```
899 REM **OPTION 5**
900 PRINT@64,“ENTER A NUMBER TO
    LIST EVERYONE WITH AT LEAST
    THAT MANY PETS”
910 INPUT A
915 PRINT:PRINT
920 FOR J=1 TO R
925 FOR K=1 TO C
930 PT(J)=PT(J)+N(J,K)
935 NEXT K
940 IF PT(J)>=A THEN PRINT
    CH$(J);“□”;PT(J):FOUND=1
945 PT(J)=Ø
950 NEXT J
955 IF FOUND=Ø THEN PRINT:PRINT
    “NO-ONE HAS”;A;“OR
    MORE PETS”
960 PRINT:PRINT“PRESS A KEY TO
```

```
   RETURN TO MENU”
970 IF INKEY$=“” THEN 970
980 RETURN
```

Q+A

What are the problems of nested loops?

Setting up an array gives you good practice with IF . . . THEN statements and FOR . . . NEXT loops. (See how many you can count in the pets survey program—there are quite a lot.) But the trickiest parts of an array are the nested FOR . . . NEXT loops as it is very easy to get the rows and columns mixed up.

Have a look at the lines that read in the data for the array. These are in the first part of the program on pages 27Ø and 271.

Each line of DATA corresponds to a row of the array, so the loop to read in the rows is the outside one.

If the line of DATA held a column from the array then the loop for the columns would be the outside one.

This is so whether the array is written as N(R,C) or N(C,R). The order of the subscripts C and R doesn't matter at all except that it must be consistent throughout the program.

The headings for the rows and columns are set up with separate loops so you shouldn't have any trouble with these.

Before the computer can compare the number you type in with the totals for each child, it first has to work out what those totals are. Lines 92Ø and 925 set up the loops to loop through the rows and columns of the array while Line 93Ø works out the total number of pets in each row. The totals are stored in an array called PT() or, on the Spectrum, p(). If the total for any row is greater than or equal to the number you typed in then the name of the child is printed out, along with the number of pets they own.

As soon as someone's name is printed the FOUND flag is set to one. If FOUND is still equal to zero when the computer gets to Line 955 then it means no-one has that many pets and the computer prints out a message to tell you so.

The last option—option 6—displays the array on the screen. It is rather difficult to fit in all the names of the pets across the top of the chart, so the program prints out numbers instead. It does, however, print out the names of the owners.

```
999 REM **OPTION 6**
1ØØØ PRINT '“CHILDREN”,“PETS”
1Ø1Ø PRINT 'TAB 9;
1Ø15 FOR j=1 TO c
1Ø2Ø PRINT j;“□□”;
1Ø25 NEXT j: PRINT
1Ø3Ø FOR j=1 TO r
1Ø35 PRINT 'c$(j);TAB 9;
1Ø4Ø FOR k=1 TO c
1Ø5Ø PRINT n(j,k);“□□”;
1Ø6Ø NEXT k
1Ø65 NEXT j
1Ø7Ø PRINT ''“PRESS A KEY TO RETURN
   TO MENU”
1Ø8Ø PAUSE Ø
1Ø9Ø RETURN
```

```
999 REM ## OPTION 6##
1ØØØ PRINT “CHILDREN”,,“PETS▣”
1Ø1Ø PRINT TAB(6);
1Ø15 FOR J=1 TO C
1Ø2Ø PRINT J;SPC(2);
1Ø25 NEXT J:PRINT
1Ø3Ø FOR J=1 TO R
1Ø35 PRINT CH$(J);TAB(6);
1Ø4Ø FOR K=1 TO C
1Ø5Ø PRINT N(J,K);SPC(2);
1Ø6Ø NEXT K:PRINT:NEXT J
1Ø7Ø PRINT“▣PRESS A KEY TO
   RETURN TO MENU”
1Ø8Ø GET A$:IF A$=“” THEN 1Ø8Ø
1Ø9Ø RETURN
```

You'll need to make two changes to the Commodore 64 program to allow the display to fit on the Vic 20 screen:

```
1Ø2Ø PRINT J;
1Ø5Ø PRINT N(J,K);
```

The routine starts out by printing the main headings “CHILDREN” and “PETS”, then numbers the pets from 1 to 7. (You'll have to change the spacing in Lines 1Ø2Ø and 1Ø5Ø if you have more or less than 7 columns in your own survey.) The next few lines print out the array a row at a time, with the name of the child at the left then the number of pets in each of the columns.

```
999 REM**OPTION 6**
1ØØØ PRINT'“CHILDREN”,
   “PETS”
1Ø1Ø PRINT'TAB(9);
1Ø15 FOR J=1 TO C
1Ø2Ø PRINT;J;SPC(4);
1Ø25 NEXT
1Ø3Ø FOR J=1 TO R
1Ø35 PRINT' CH$(J);TAB(9);
1Ø4Ø FOR K=1 TO C
1Ø5Ø PRINT;N(J,K);SPC(4);
1Ø6Ø NEXT K,J
1Ø7Ø PRINT''“PRESS A KEY TO RETURN
   TO MENU”
1Ø8Ø A=GET
1Ø9Ø RETURN
```

```
999 REM **OPTION 6**
1ØØØ PRINT@32,“CHILDREN”,“PETS”
1Ø1Ø PRINT@72,;
1Ø15 FOR J=1 TO C
1Ø2Ø PRINTJ;
1Ø25 NEXT:PRINT
1Ø3Ø FOR J=1 TO R
1Ø35 PRINT:PRINT CH$(J)TAB(8);
1Ø4Ø FOR K=1 TO C
1Ø5Ø PRINT N(J,K);
1Ø6Ø NEXT K,J
1Ø7Ø PRINT:PRINT:PRINT“PRESS ANY
   KEY TO RETURN TO MENU”
1Ø8Ø IF INKEY$=“” THEN 1Ø8Ø
1Ø9Ø RETURN
```

ADAPTING THE ARRAY

An array used like this is sometimes called a database. You can create a database for all sorts of subjects—you could use one for a wildlife survey for example. This could show animals and birds you spotted and the type of habitat you found them in. The names of the animals would go along one side of the array and the habitats along the other. If you then filled in an array at different times of the year or in different parts of the country you could see how the distribution of the species varied.

Another use would be for a traffic survey, to see which vehicles used a road at different times of the day. Or you could set up a foods database showing how much fats, carbohydrate, protein and calories were in different types of foods. With this one you could print out all foods containing more than a certain amount of protein or less than so many calories, or the composition of a particular food.

A more immediate use of arrays is to keep track of stock in a warehouse or shop. In this case the rows and columns of the array refer to actual shelves and positions in a storeroom, and the data relates to the numbers of each item in stock. The computer program can then be designed to update the records as items are sold or stock replaced and it can print out reorder messages when numbers fall below a certain minimum amount.

You're sure to be able to think up a use to suit your own hobby, business or interest.

HOW TO ENTER MACHINE CODE

- FINDING A PLACE FOR MACHINE CODE
- MACHINE CODE MONITORS
- USING THE MONITOR
- RUNNING A PROGRAM

When you switch on your machine it automatically goes into BASIC. So you must use a BASIC program to POKE in your machine code program and a BASIC command to run it

When you enter the wonderful world of machine code programming you run into a number of strange paradoxes. Although machine code is the language the microprocessor itself uses, you cannot feed it directly into your home computer. It must be entered by another program—which on your micro means using BASIC. Nor can you simply run a machine code program using a machine code command. You have to call it out of a BASIC program or by using a BASIC instruction.

To enter a machine code routine, you have to POKE it byte-by-byte into the computer's memory. Ironically, on the Spectrum, ZX81 and Commodores you cannot POKE a hex number, only a decimal one. So the hex numbers of your machine code routine have to be converted back into decimal before they can be entered.

The Dragon and Tandy allow you to POKE hex numbers, so this problem does not arise. And as the BBC Micro and the Electron have built-in assemblers, Acorn owners will have no interest in entering machine code and can skip direct to forthcoming chapters on assembly language.

WHERE TO PUT IT

Before you enter a machine code program, you have to decide where you are going to put it. This is a very important task. Obviously you must not put it in an area that the computer itself uses, otherwise your program will either be overwritten by fresh data from ROM, or the computer will fail to function because you have overwritten vital data.

You must be wary when using the BASIC area too. As your machine code routine may be called out of a BASIC program, your machine code could be overwritten while the BASIC program is RUNning, causing the whole thing to crash.

With small programs you can use areas like the cassette buffer, provided you are not going to use the cassette player.

In the Spectrum you can use the user-defined graphics area—between FF58 and FFFF in the 48K model—provided no UDGs are being used. But usually, machine code programs are tucked between the UDG area and the end of the BASIC area by moving RAMTOP down the memory.

This way the BASIC area is squeezed, but as it can't extend past RAMTOP, the machine code is protected from overwriting.

RAMTOP is a system variable and its position is given by the pointer in memory locations 5CB2 and 5CB3. To move the RAMTOP down you could just POKE a lower value into these two locations. But the Spectrum has another BASIC command, CLEAR. This does essentially the same thing and should be followed by the decimal address of the place you want the RAMTOP dropped to.

CLEAR clears the display file—like a CLS does—and also all the variables. It resets the PLOT position to Ø, Ø—that is the bottom left, restores the DATA pointer to the start, clears out the GOSUB stack and places it under the new value for RAMTOP. On a 16K Spectrum the usual command is:

```
CLEAR 31999
```

This moves RAMTOP down to 31,999 or 7CFF hex, leaving the 600 bytes above it—that is up to the end of the UDG area at 32,6ØØ—protected for machine code programs.

On the 48K model, the usual command is:

```
CLEAR 63999
```

Again this drops RAMTOP so that, this time, the BASIC area finishes at 63,999 or F9999 in hex and means that you can start your machine code program at 64,000 or FAØØ.

For most purposes this will leave more than enough room for your machine code programs. In fact you needn't leave quite so much room, but it is always better to leave a little more space than you need in case you want to amend or expand your program later.

With longer machine code programs you may have to drop RAMTOP further. You can CLEAR all the way down to 23,821. But in practise doing this would be no good to you because with RAMTOP so low there is no room even to enter a line of BASIC.

On the ZX81 it is possible to drop RAMTOP by POKEing memory locations 16,388 and 16,389 to give a protected area where your machine code programs cannot be overwritten. But the problem with this approach is that you can't SAVE programs in that area.

Short machine code programs—less than 32 bytes—can be POKEd into the printer buffer, which occupies memory locations 16,444 to 16,476, provided no printer is being used. There are other small areas which you can POKE your machine code programs into but, again, you won't be able to SAVE them. On the ZX81 you can only SAVE programs in the BASIC program area.

There are three ways you can put your machine code program into the BASIC area and still protect them against overwriting. You can enter them as REM statements, arrays or character strings. In general you'll find that the first method is the best.

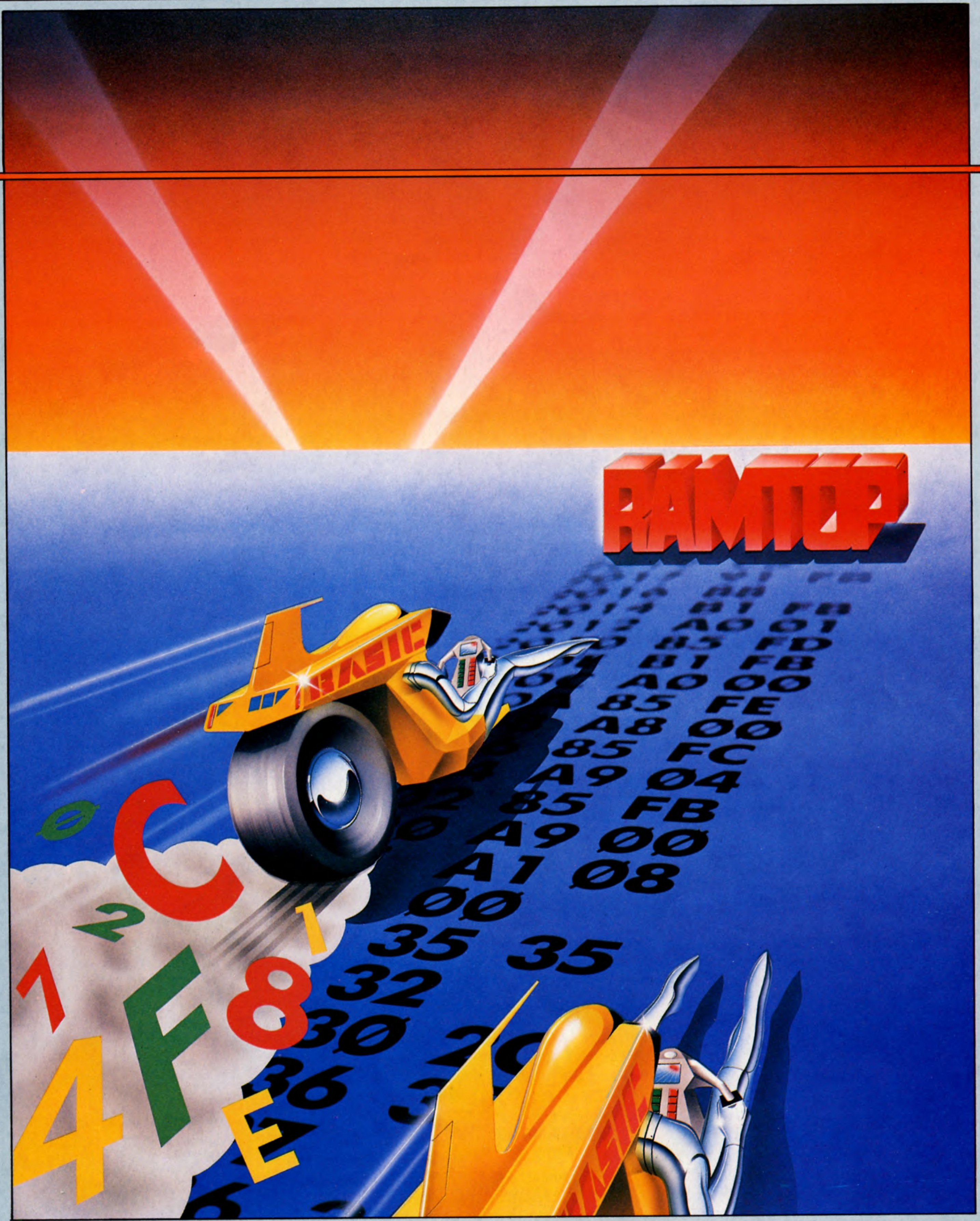
To enter a machine code program as a REM statement you must work out how long it is—that is, count the bytes. Then on the first line of your BASIC program type REM followed by a number of full stops, say—the number of full stops must be at least equal to the number of bytes in your machine code program. In fact you don't have to choose full stops, as any character will do to fill out the REM statement—but it is best to keep them all the same so that you can spot the end of your machine code program in memory.

You then POKE your machine code program into the memory locations occupied by the REM statement, a byte at a time. If your REM statement is the first line of your BASIC program the area cleared begins at memory location 16,514.

Another way to create protected space is to dimension an array. The BASIC line:

```
1Ø DIM A(1ØØ)
```

will give 500 free memory locations in the variables area—for each element of the array the ZX81 leaves five free locations to store floating point numbers. To find out where you


RAMTOP
BASIC
B1 FB
AO Ø1
85 FD
B1 FB
AO ØØ
85 FE
A8 ØØ
85 FC
A9 Ø4
85 FB
A9 ØØ
A1 Ø8
ØØ
35 35
32
3Ø
36
BASIC
C
F
4
8
E
Ø
2
1
7

should begin storing your machine code in this case you have to PEEK at the system variable VARS, which is located in 16,400 and 16,401, with a direct mode command like:

```
PRINT PEEK 164ØØ + PEEK 164Ø1 + 6
```

This gives the start address of the protected area you have created—the extra six memory locations contain details of the array. You should POKE your machine code program in byte by byte from here.

Another variation is to create space in a string with a BASIC line like:

```
1Ø LET S$ = "................"
```

with more full stops than the number of bytes in the machine code program. Again, this protected space appears at the beginning of the variables area in memory. And to find the start address you PEEK VARS and add six. The six extra memory locations in this case carry the name of the string.

The problem with these two last methods is that the variables area is CLEARed if you hit the CLEAR key or RUN a BASIC program, and you will lose your machine code program.

On the Commodore 64, the memory area from CØØØ to CFFF in hex, or 49,152 to 53,247 decimal, is reserved for machine code and normally this area is as much as you will need for your machine code routines. The Vic 20 has no reserved area.

Small programs—less than 191 bytes long—can be stored in the tape buffer on the 64 and the Vic, provided you're not intending to use your cassette recorder. The tape buffer is located between Ø33C to Ø3FB in hex, or 828 to 1,019 in decimal.

But if more space is required you can limit the BASIC area and leave a protected area above it—the same way the CLEAR command works on the other machines. On the Commodore 64 this is performed by POKEing 52 and 56 with, say, 32. This clears down to 8,192 which gives 32,768 bytes that cannot be overwritten by BASIC for machine code routines, and leaves around 6,144 bytes for BASIC programs.

After you have done this you must type in CLR—that is type in the letters C, L and R separately. Don't just press the CLR key or any program you have in the BASIC area is liable to become confused.

Memory locations 52 and 56 are the high bytes of the two points that specify the top of the BASIC area. The low bytes, 51 and 55, are usually Ø. So 52 and 56 specify which page BASIC ends on. And if you POKE them with 32, you shift the pointer to 32 × 256, or 8,192.

You can close the BASIC area down completely by POKEing 52 and 56 with 8—this takes the top of BASIC down to 2,048 which is the bottom of the BASIC area. You can even POKE 52 and 56 with Ø! But in either case the Commodore will then be no good to you as you won't be able to enter any BASIC and thus it would be impossible to call any machine code routines.

You can also move the bottom of the BASIC area up and protect your machine code routines by tucking them in underneath. This is done by POKEing memory locations 43 and 44. These are the low and high bytes of the pointers that specify the bottom of the BASIC area. Again you could move up the bottom of BASIC to 8,192 by POKEing location 43 with 1 and location 44 with 32. You'll also have to POKE the new start of BASIC—location 8192—with zero. This will leave plenty of room for machine code routines below and BASIC above.

After you have done this you must type in NEW, otherwise any program you have in the BASIC area will become confused and crash.

On the Vic, the amount you can lower RAMTOP depends on how much the memory has been expanded. PEEK 44 to see where the bottom of the BASIC area is, then POKE 52 and 56 with that number, plus 1 or 2, to allow space for a little bit of BASIC.

The easiest way to protect machine code programs is to use the computer's CLEAR command, which moves down RAMTOP and leaves a space above it for your machine code program which will be protected from overwriting in BASIC. It also sets all numeric variables to Ø and string variables to null strings. The command usually takes the form:

```
CLEAR 2ØØ,3ØØØØ
```

This moves RAMTOP down from &H7FFF, or 32,767 in decimal, to 30,000, leaving 2,767 bytes free for machine code programs.

On these computers the CLEAR command has another, separate, unrelated function—the assigning of string space. How much string space is reserved is given by the first number after the CLEAR. The computer automatically sets aside 200 bytes under RAMTOP for strings when you turn the machine on. This is sufficient for most purposes, so when you artificially lower RAMTOP you might as well leave the same amount—200 bytes—below it for strings.

You can drop RAMTOP almost as far as the top of the graphics area—though the machine will need a little BASIC space to execute the command. On the Dragon and Tandy the top of the graphics area itself can be shifted using the instruction PCLEAR.

PCLEAR 1 will leave only one of the graphics pages—so you can CLEAR down to &HC4Ø. say, PCLEAR 8 assigns all eight graphics pages so you would only be able to clear down to &H368Ø. If you don't use the command CLEAR, the computer will automatically assign four graphics pages, so you will be able to CLEAR down as far as &H1E8Ø.

These limits depend very much on what is in the machine at the time. If there is any sort of BASIC program on board, you are likely to get an out-of memory error message. So for all practical purposes, working this close to the limit is no good to you, as you would have to enter a BASIC program at some point from which to CALL your machine code program.

It is unlikely that you have to CLEAR further than 30,000. This will give you, 2,767 bytes free. If you want to start your machine code program at a nice round number like 30,000, you should type in:

```
CLEAR 2ØØ,29999
```

The second number, following the comma, after the word CLEAR specifies the highest address available to BASIC programs, so the locations safe for machine code programs start from the one above.

MACHINE CODE MONITORS

The following programs allow you to key in machine code, SAVE machine code programs on tape and examine them in memory. The appropriate CLEAR command must be given before you enter the program. Once you have done that, entered the program and RUN it you will be asked to supply a start address. With the Spectrum , ZX81, Tandy and Dragon this should be the memory address above the one you have CLEARed down to. Remember that the CLEAR command specifies the last address of BASIC, and the location above it is where your machine code program should begin.

With the Commodore you should begin your program at CØØØ, unless your program is more than 4,095 bytes. In that case you should put it in the BASIC area by POKEing 52 and 56 with 32 and start your machine code routine at 8,192 in decimal. For the Vic 20, where you start your machine code program depends on how much your machine has been expanded (see above). The Commodore does not need the difference of 1 between the value of the pointers marking the end of BASIC and the start of your machine code routines. The pointers you POKE at 52 and 56 already take that into account.

Now key in the machine code monitor for your machine:

2017 91 FB
2016 88
2014 B1 FB
2012 A0 01
2010 85 FD
200E B1 FB
200C A0 00
200A 85 FE
2008 A8 00
RAMTOP
1FF7 2A
1FF6 4F
1FF5 4B
1FF2 2E 4D 4B
1FEF 2C 2F 2C
4B 3A
4C 3A
FE4 4B
FE1 4C 4C
FE0 4B
BASIC

```
5 POKE 23658,8
10 PRINT INVERSE 1;AT 5,6;
   "□SPECTRUM MONITOR□"
15 PRINT AT 8,4;"1: – □ENTER□
   MACHINE CODE"
20 PRINT AT 10,4;"2: – □EXAMINE
   □MEMORY"
30 PRINT AT 12,4;"3: – □SAVE□
   BYTES□TO□TAPE"
70 LET A$ = INKEY$: IF A$ < "1" OR
   A$ > "3" THEN GOTO 70
80 CLS : GOSUB 100 + 200*(VAL
   A$ – 1)
90 RUN
100 INPUT "START ADDRESS?□";SA
110 INPUT LINE D$
120 IF D$ = "" THEN GOTO 110
125 IF D$(1) = "#" THEN RETURN
127 IF LEN D$ = 1 THEN GOTO 110
130 LET S$ = D$(1 TO 2)
140 FOR N = 1 TO 2
150 IF S$(N) > "9" THEN LET S$(N)
   = CHR$ (CODE S$(N) – 7)
155 IF S$(N) > "?" OR S$(N) < "Ø"
   THEN PRINT FLASH 1;"INVALID□
   CHARACTER": GOTO 110
160 NEXT N
170 POKE SA, (CODE S$(1) – 48)*16
   + CODE S$(2) – 48
180 PRINT SA,D$( TO 2)
190 LET D$ = D$(3 TO )
200 LET SA = SA + 1
210 GOTO 120
300 INPUT "START□ADDRESS?□";SA
310 INPUT "PRINTER□(Y□OR□N)?□";
   LINE P$
320 LET ST = 2: IF P$ = "Y" THEN
   LET ST = 3
330 PRINT #ST;SA;
340 FOR M = Ø TO 7
350 LET H$ = "##"
360 LET H$(1) = CHR$ (INT (PEEK
   (SA + M)/16) + 48)
370 LET H$(2) = CHR$ (48 + PEEK
   (SA + M) – 16*(CODE H$(1) – 48))
380 FOR N = 1 TO 2
390 IF H$(N) > "9" THEN LET H$(N)
   = CHR$ (CODE H$(N) + 7)
400 NEXT N
410 PRINT #ST;TAB 7 + 3*M;H$;
420 NEXT M
440 LET SA = SA + 8
445 PRINT #ST: POKE 23692,Ø
450 LET A$ = INKEY$: IF A$ = ""
   THEN GOTO 450
460 IF A$ = CHR$ 13 THEN RETURN
470 GOTO 330
500 INPUT "START ADDRESS?□";SA
510 INPUT "NUMBER OF BYTES?□";N
520 INPUT "FILE NAME?□"; LINE N$
530 IF LEN N$ < 1 OR LEN N$ > 10
   THEN GOTO 520
540 SAVE N$CODE SA,N
550 RETURN
```

Before RUNning this program you must type in a REM statement on Line 1 followed by the appropriate number of characters to contain your machine code program—one character per byte of machine code:

```
10 PRINT AT 4,9;"ZX81 MONITOR"
15 PRINT AT 8,4;"1:—ENTER MACHINE
   CODE"
20 PRINT AT 10,4;"2:—EXAMINE MEMORY"
50 LET A$ = INKEY$
60 IF A$ < "1" OR A$ > "2" THEN GOTO 50
70 CLS
80 GOSUB 100 + 200*(VAL A$ – 1)
85 CLS
90 RUN
100 PRINT AT 21,Ø;"START ADDRESS?"
105 INPUT SA
110 INPUT D$
120 IF D$ = "" THEN GOTO 110
125 IF D$(1) = "$" THEN RETURN
130 LET S$ = D$(1 TO 2)
135 SCROLL
140 FOR N = 1 TO 2
150 IF S$(N) < "Ø" OR S$(N) > "F" THEN
   PRINT "INVALID CHARACTER"
155 IF S$(N) < "Ø" OR S$(N) > "F" THEN
   GOTO 110
160 NEXT N
170 POKE SA, (CODE S$(1) – 28)*16 + CODE
   S$(2) – 28
175 SCROLL
180 PRINT SA,D$(TO 2)
190 LET D$ = D$(3 TO)
195 IF LEN D$ = 1 THEN GOTO 110
200 LET SA = SA + 1
210 GOTO 120
300 PRINT AT 21,Ø;"START ADDRESS?"
310 INPUT SA
320 SCROLL
330 PRINT SA;
340 FOR M = Ø TO 7
350 LET H$ = "□□"
360 LET H$(1) = CHR$ (INT (PEEK
   (SA + M)/16) + 28)
370 LET H$(2) = CHR$ (28 + PEEK
   (SA + M) – 16*(CODE H$(1) – 28))
400 PRINT TAB 7 + 3*M;H$;
410 NEXT M
440 LET SA = SA + 8
450 LET A$ = INKEY$
460 IF A$ = "" THEN GOTO 450
470 IF A$ = CHR$ 118 THEN RETURN
480 GOTO 320
```

```
5 POKE 53280,Ø:POKE 53281,Ø:HH$ =
   "Ø123456789ABCDEF"
10 PRINT "♥▣▣▣▣▣▣▣◢"
   TAB(8)"1: – □ENTER□MACHINE□
   CODE"
20 PRINT TAB(8)"▣▣2: – □□□
   EXAMINE□MEMORY"
30 PRINT TAB(8)"▣▣3: – □SAVE
   □BYTES□TO□TAPE"
40 GET A$:IF A$ < "1" OR A$ > "3"
   THEN 40
50 ON VAL(A$) GOSUB 100,200,300
60 GOTO 10
100 INPUT "♥START ADDRESS□";SA
110 INPUT "⊟▣▣□□□□□
   ▌▌▌▌▌▌▌▌▌▌";D$:IF D$ = "" THEN 110
120 IF D$ = "#" THEN RETURN
125 W = Ø:FOR Z = 1 TO 16:IF LEFT$(D$,
   1) = MID$(HH$,Z,1) THEN W = W + 1
126 IF RIGHT$(D$,1) = MID$(HH$,Z,1)
   THEN W = W + 1
127 NEXT:IF W < 2 THEN 110
130 A = ASC(D$) – 48:B = ASC(RIGHT$
   (D$,1)) – 48
140 C = B + 7*(B > 9) – (LEN(D$) = 2)*(16
   *(A + 7*(A > 9)))
150 POKE SA,C:PRINT "⊟▣▣▣▣▣
   □□□□□□□□□□□□□□□
   □□□□□□□□□□□□□□□
   □□□□□□□□□□◙"SA,D$
160 SA = SA + 1:GOTO 110
200 INPUT "♥START□ADDRESS□";SA
210 INPUT "PRINTER□(Y□OR□N)□";
   P$:PRINT "♥"
220 IF P$ = "Y" THEN OPEN 4,4:CMD4
230 PRINT SA;:FOR M = Ø TO 7
240 A = (PEEK(SA)/16):PRINT MID$
   (HH$,A + 1,1);MID$(HH$,PEEK(SA)
   – INT(A)*16 + 1,1)"□";
250 SA = SA + 1:NEXT
260 IF P$ = "Y" THEN PRINT #4,"";:
   CLOSE4
270 GET A$:IF A$ = "" THEN 270
280 IF A$ = CHR$(13) THEN RETURN
290 PRINT:GOTO 220
300 CLR:INPUT "♥ENTER□START□
   ADDRESS";A:A = A – 3:AA = INT(A/256):
   A2 = A – AA*256
310 INPUT "ENTER□END□ADDRESS□□";
   B:B = B + 1:BB = INT(B/256):B2 = B
   – BB*256
315 INPUT "INPUT□FILE□NAME□□
   □□";N$
320 PRINT"♥P▛44,"AA":P▛43,"A2
330 PRINT"▣▣P▛46,"BB":P▛45,"
   B2:PRINT"▣▣SAVE"CHR$(34)
   N$CHR$(34)
340 PRINT"▣▣▣▣▣▣▣▣▣
   P▛44,"PEEK(44)":P▛43,"PEEK(43)
```

```
350 PRINT"▣▣P▛46,"PEEK(46)
   ":P▛45,"PEEK(45):PRINT
   "▣▣RUN⊟"
```

You'll need to change these Lines to allow the Commodore 64 program to work on the Vic 20:

```
5 POKE 36879,8:HH$="Ø123456789
   ABCDEF"
10 PRINT "♡▣▣▣▣▣▣▣◢
   1:—□ENTER MACHINE CODE"
20 PRINT "▣▣2:—□□□EXAMINE
   MEMORY"
30 PRINT "▣▣3:—□SAVE BYTES TO
   TAPE"
150 POKE SA,C:PRINT "⊟▣▣▣
   ▣▣□□□□□□□□□□□
   □□□□□□□□□□□□□
   ○"SA,D$
230 PRINT SA;:FOR M=Ø TO 4
```

```
10 CLS
20 PRINT@196,"1: — □ENTER□MACHINE
   □CODE"
30 PRINT@260,"2: — □EXAMINE□
   MEMORY"
40 PRINT@324,"3: — □SAVE□BYTES
   □TO□TAPE"
50 A$=INKEY$:IF A$<"1" OR A$>"3"
   THEN GOTO 50
60 CLS
70 ON VAL(A$) GOSUB 200,400,600
80 GOTO 10
200 INPUT "START□ADDRESS□";SA
210 LINE INPUT D$
220 IF D$="" THEN GOTO 210
230 IF LEFT$(D$,1)="#" THEN
   RETURN
240 S$=LEFT$(D$,2)
250 POKE SA,VAL("&H"+S$)
260 PRINTSA,LEFT$(D$,2)
270 D$=MID$(D$,3)
280 SA=SA+1:GOTO 220
400 INPUT "START□ADDRESS□";SA
410 PRINT"OUTPUT□TO□PRINTER□
   (Y/N)□?"
420 A$=INKEY$:IF A$="" THEN 420
430 P=Ø:IF A$="Y" THEN P=-2
440 PRINT #P,SA;
450 FOR M=Ø TO 7
460 PRINT #P,"□";RIGHT$("□"+
   HEX$(PEEK(SA+M)),2);
470 NEXT:PRINT #P
480 SA=SA+8
490 A$=INKEY$:IF A$="" THEN 490
500 IF A$=CHR$(13) THEN RETURN
510 GOTO 440
600 INPUT "START□ADDRESS□";SA
610 INPUT "NUMBER□OF□BYTES□";N
620 LINE INPUT"FILE NAME□";N$
630 CSAVEM N$,SA,SA+N,SA
640 RETURN
```

HOW THE MONITOR WORKS

When you RUN the program appropriate for your machine, it will display a menu. This asks you whether you want to 'Enter Machine Code', 'Examine Memory' or—except on the ZX81—'Save Bytes to Tape'.

If you want to enter machine code you press 1. When you have given the machine a start address and pressed ENTER or RETURN, start feeding in your machine code. This consists of pairs of hex digits and you must feed the digits in two at a time.

You can enter as many—or as few—pairs as you wish before you press ENTER or RETURN. But it is best to type in a line at a time and check the digits rigorously before you ENTER them. It is much easier to edit them while they are still on the screen than when they are in memory. On the Commodore the hex is entered two digits at a time.

You will notice that the programs for the Spectrum, ZX81 and Commodores carefully translate your hex numbers back into decimal before POKEing them into memory. This is because the POKE command is a BASIC statement (see pages 240 to 247) and the BASIC on these two machines does not accept hex. Still, it is best for you to think in hex as it gives you more of a feel for the way the machine is working.

You must end your machine code programs with a hash # sign—or a dollar sign on the ZX81. This returns you to the menu.

If you select option 2 to examine the memory, the program will ask you again for a start address—this time it is the start of the memory area that you want to look at. To examine the whole of your machine code program, this start address should be the same as the start address for the whole of the program you've given before.

Next, except on the ZX81, you're asked whether you want to PRINT out a hard copy of the machine code program on your printer, if you have one. If you don't want to do this, and press N, the program will print out the start address and the contents of that location and the subsequent seven memory locations in one line across the screen.

If you press any key on the keyboard—except ENTER or RETURN—the address of the location eight on from the start address will be printed, along with its contents and the contents of the seven subsequent bytes. This appears on a separate line underneath and you can PRINT out the whole of your machine code program line-by-line in this fashion.

If you spot an error and want to correct it, press ENTER or RETURN and the program will return you the menu. Press 1 to enter machine code and the computer asks you for a start address again. This time you give it the address of the byte that is wrong. If it's a whole series that is wrong, just give it the address of the first byte of the series. You then enter the correct bytes, one after the other again.

If there is only a single error, though, you could correct it by BREAKing out of the monitor program and POKEing the appropriate location with the correction. Remember, though, on the Spectrum and the Commodore you have to POKE decimal number.

When you've finished entering the correction, you press ENTER or RETURN, then examine the memory again to make sure that you've got it right this time.

SAVING YOUR ROUTINES

If you want to keep this machine code monitor, you should SAVE it separately from your machine code programs on tape in the usual way (see page 23). The 'Save Bytes to Tape' option in the program itself only saves the machine code routines that have been entered with this program.

On the ZX81, though, you must SAVE your machine code routines along with your monitor in the normal way.

On the other machines if you press 3, the program will ask you for a start address once more. This should be the start address of the machine code routine you've just fed in, though you can save any part of the memory by using a different start address.

The Spectrum, Dragon and Tandy versions of this program then ask you for the numbers of bytes your machine code routine takes up. You can work this out by counting the pairs of hex digits. Each pair is a byte. Then you must enter a name for the routine—so that your computer will be able to identify it when you want to LOAD it back from tape later.

The routine is then SAVEd by pressing the play and record buttons on your cassette player in the normal way. When that is completed, the menu comes up again.

The Commodore version asks for an end address. This is the start address, plus the number of bytes the program takes. Or you can work out the end address by examining the memory.

Once you have given your routine a name, the machine will display a program line at the top of the screen. HOME the cursor here and press RETURN three times. You then press the play and record buttons on your cassette player to SAVE the program in the normal way.

But the Commodore machine code monitor cannot be used again until the pointers that were changed during the SAVE routine have been set again. To do that, take the cursor to the beginning of the program line half way down the screen and press RETURN three times. This will POKE the original values back into the pointers and RUN the program again for you automatically.

LOADING MACHINE CODE

With the Spectrum, Dragon and Tandy you simply LOAD the machine code routine in the normal way (see page 22), though if you have the machine code monitor program in the machine's memory and it is RUNning, you'll have to press BREAK to escape from the program first.

Remember to repeat the CLEAR command so that your machine code is protected if you have switched the machine off or otherwise altered the position of RAMTOP since you SAVEd your machine code routine.

With the Commodores you have to use a more specialized LOAD command when entering machine code from tape to make sure that it is put back in the same place it was copied from when you SAVEd it. The LOAD command has the following format:

```
LOAD "Routinename",1,1
```

"*Routinename*" is the name of the machine code routine you are aiming to LOAD. This is always in quotes. The first 1 is the device number, 1 means cassette recorder. The second 1 tells your Commodore to replace the program in exactly the same spot in memory it came from.

If you are LOADing from disk, you should suffix the LOAD command with ,8,1.

Again, if you had altered any of the BASIC pointers when you wrote the machine code routine and SAVEd it, you must set them again to the same values if they have been altered for any reason since then—like you switched the machine off. Otherwise your machine code routine will not be protected and may be overwritten by BASIC.

With the ZX81, your machine code program can be LOADed off tape along with the monitor in the usual way.

SCROLLING BACKWARDS

The following routines will scroll the screen backwards—that is from top to bottom rather than from bottom to top. Enter the appropriate routine for your machine using the machine code monitor. Remember not to type in any spaces between the numbers on the Spectrum, ZX81, Dragon or Tandy. On the Commodores the numbers are entered one at a time.

```
21 9F 58 11 BF 58 Ø6 Ø8
25 15 E5 D5 C5 Ø1 AØ ØØ
ED B8 Ø6 Ø2 C5 D5 11 ØØ
F9 19 D1 Ø1 2Ø ØØ ED B8
E5 21 ØØ F9 19 EB E1 Ø1
EØ ØØ ED B8 C1 1Ø E5 C1
D1 E1 1Ø D4 21 ØØ 3F Ø6
Ø8 C5 24 E5 AF 77 54 5D
13 Ø1 1F ØØ ED BØ E1 C1
1Ø EF 21 9F 5A 11 BF 5A
Ø1 AØ Ø2 ED B8 21 ØØ 58
11 Ø1 58 3A 8D 5C 77 Ø1
1F ØØ ED BØ C9 #
```

```
Ø1 18 Ø3 2A ØC 4Ø Ø9 54
5D Ø1 F7 Ø2 2A ØC 4Ø Ø9
ED B8 2A ØC 4Ø 23 36 ØØ
54 5D 13 Ø1 1F ØØ ED BØ
C9
```

```
A9 13 2Ø D2 FF A9 11 2Ø
D2 FF A9 9D 2Ø D2 FF A9
94 2Ø D2 FF A9 AØ 85 DA
A9 ØD 2Ø D2 FF 6Ø #
```

```
8E Ø5 EØ A6 84 A7 89 FE
ØØ 3Ø 88 EØ A6 84 A7 88
2Ø 8C Ø4 ØØ 2C F3 3Ø 89
Ø2 Ø1 8C Ø6 ØØ 25 E4 39
#
```

RUNNING THE PROGRAMS

Once you have keyed in your machine code program and checked that there are no errors in it by examining it in memory, you will want to run it. But it will not respond to the usual RUN command as that is a BASIC instruction and will only RUN BASIC programs. Special instructions are used to run machine code programs.

The Spectrum and ZX81 use the BASIC keyword USR, followed by the start address of the machine code program. USR returns the value of the BC register once the machine code routine has been completed. In itself this is not very useful, but it does mean that the machine code program does have to run while it is being worked out.

Unfortunately, USR is not a command but a function. And the structure of Sinclair BASIC demands that to be executed, a line must start with a command. So USR must be prefaced by a command word. On the Spectrum use:

```
RANDOMIZE USR 32ØØØ
```

and on the ZX81:

```
RAND USR 16514
```

(Note that the key marked RAND on the Spectrum gives RANDOMIZE on the screen. On the ZX81 the RAND key gives RAND on the screen.) In itself, this command is meaningless, but it does succeed in running the machine code routine.

In fact any BASIC command can be used to preface USR. PRINT USR 32ØØØ will do, though this will actually PRINT out the value to BC on the screen.

RANDOMIZE USR 32ØØØ (or RAND USR 16514) is used most commonly because it avoids such unwanted side effects. But it should be avoided if you are using random numbers anywhere in your program, as it will set the seed again. In this case on the Spectrum use:

```
LET L=USR 32ØØØ
```

and on the ZX81 use:

```
LET L=USR 16514
```

This sets the variable L to the value of the BC register when the machine code program has finished running—no great shakes in itself, but Sinclair users will note that LET,L,= and USR are all on the same key, which makes things easier.

The easiest way to run a machine code program on your Commodore 64 or Vic 20 is to type in SYS followed by the start address of the machine code routine. But if you want to pass variables between the machine code routine and BASIC you can use the BASIC function USR.

The usual syntax is:

A=USR(B) or PRINT USR(B) in direct mode.

USR sends the computer to the machine code routine whose start address is stored in memory locations 785 and 786 (in the usual low-byte/high-byte format). So you must POKE these locations with the start address of the machine code routine you want to run first to set the pointers.

The value of B is passed into the machine code program through the first floating point accumulator, which occupies memory locations 97 to 1Ø2 (97 is the exponent, 98 to 1Ø1 is the mantissa and 1Ø2 carries the sign). USR also returns the contents of the floating point accumulator when the machine code routine is finished. This way it can be used to pass values from your machine code routine back to the BASIC program.

On the Dragon and Tandy there are two instructions which run machine code programs. EXEC is a command which should be followed by the start address of the machine code routine you want to run. For example:

```
EXEC24ØØØ
```

will run the machine code routine which starts at memory location 24000 decimal. But the USR command allows variables to be passed between the BASIC program and the machine code routine.

First of all you must define where the machine code routine you want starts. You do this with the instruction DEFUSR. This should be followed by a number between Ø and 9, so you can define up to ten USR routines—that is you can use up to ten machine code subroutines in any BASIC program. If DEFUSR is not followed by a number, the computer assumes it to mean DEFUSRØ.

The usual syntax in a BASIC program is:

```
1Ø DEFUSR1 = 24ØØØ
```

This defines USR routine number 1 as the one that starts at memory location 24000 decimal.

The command that actually runs the machine code is USR. Due to a bug in the Dragon's ROM, to call a machine code routine, USR must be followed by Ø and the number of the routine which you have previously defined. The Tandy does not need the Ø.

The usual syntax is:

P = USRØ1 (Q) or PRINT USRØ1 (Q).

The USR command copies the value of the variable Q into the floating point accumulator. From there it can be picked up by the machine code routine.

When the machine code routine is finished the contents of the floating point accumulator are returned to the BASIC program, in this example, as variable P.

String variables can also be passed between BASIC and machine code programs in the same manner.

MEANINGFUL RELATIONSHIPS

Computers can perform operations of logic many millions of times a second when running. But logic is also a fundamental part of any program . . .

Three types of expression are used in BASIC programming. Much of the 'meat' of any program is formed by arithmetic and string expressions, but the third type, *logical* expressions, often do the real work of decision-making in a program.

Their simple function is to evaluate whether something is 'true' or 'false'. The program words and symbols used to do this are called *operators* (or, by some, *connectors* or connectives).

Two types of operator are used in logical expressions: *relational operators* (which include mathematical symbols such as <, > and =) and *logical operators* included as part of the keyword list of all forms of BASIC: AND, OR and NOT.

MORE, EQUAL, LESS

The relational operators can be used in the simplest of BASIC programs. The comparisons possible using these are:

A>B ...A is greater than B
A<B ... A is less than B
A>=B ... A is greater than or equal to B
A<=B ... A is less than or equal to B
A=B ... A is equal to B
A<>B ... A is not equal to B

You're bound to have seen these in use in dozens of programs in *INPUT* or elsewhere, and although they can be used in straightforward arithmetic it's in conjunction with IF ... THEN that their true value in conditional tests becomes evident. A typical example is:

```
IF A>B THEN PRINT "A IS GREATER
THAN B"
```

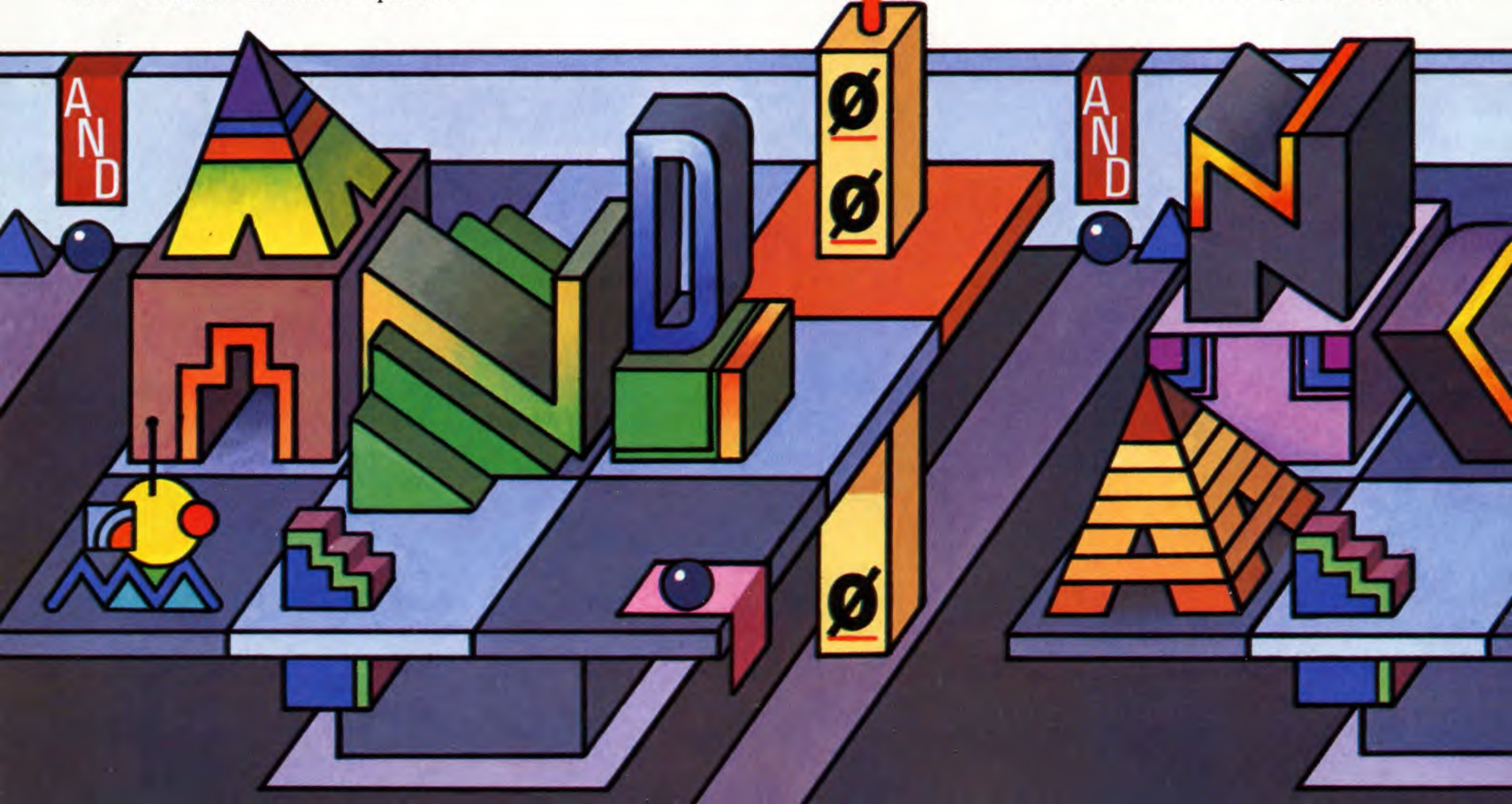
The value of A here *must* exceed that of B before the remainder of the line is executed. In any instance where the value of A remained below B, the program would simply continue on to the next line. So you can see that some measure of conditional branching is possible: if one set of values applies, the program does one thing; if another set of values applies the program does something else.

Relational operators are not limited to numerical values alone, but may also be used to compare strings. However, this must be done with caution or it may lead to some odd results. The first point to remember is that the comparison always stops with the number of characters in the shorter of the two strings being compared. So if one string contains eleven characters while the other contains seven, only the first seven of the longer string are compared with their counterparts in the shorter string.

The comparison is actually made one character at a time from left to right and this is where the odd results may come in. In a *numerical* comparison the statement 5>1Ø is obviously untrue, but in a *string* comparison you may have a statement where two variables are compared in the form A$>B$. If A$="5" and B$="1Ø" the computer first compares the left hand 'character' on each side—5 and 1—and then stops because as far as it's concerned there's nothing else to compare.

Incorrectly, a 'true' condition now .exists because 5 *is* greater than 1, but the computer ignores important remaining figures on the longer side. You have to watch out for errors such as these.

In 'real' character strings, the letters of the alphabet have the following order "A"<"B"<"C"<"D" (and so on) in com-



- RELATIONAL OPERATORS
- DECIDING IF SOMETHING IS MORE THAN, LESS THAN OR EQUAL TO SOMETHING ELSE
- LOGICAL OPERATORS
- USING **AND**, **OR** AND **NOT**
- MEANING OF TRUTH TABLES
- USING **EOR** ON THE ACORNS
- BITWISE OPERATIONS

parisons. But once again, you have to use some care because the computer doesn't actually understand what these characters mean. What it does is to compare the *character codes* of the letters (a later article explains these codes more fully)—so spaces and other non-letter characters (which also have a code value) in the string are significant. This could cause havoc in a program designed to sort strings into alphabetic order.

If each string has the same sequence of characters, then the longer string is considered numerically larger. But note that a shorter string is considered the larger in an expression such as "ABD" > "ABCD" for the comparison stops at the first difference—when the code values of "D" and "C" are compared. It works in fact in a similar way to the alphabetical priority in a dictionary or index, where 'ant' comes before 'antelope' but after 'aardvark'.

TRUE OR FALSE

After any comparison you get a result—an integer in fact. The result is Ø if the comparison is false, and (depending on your machine) −1 or 1 if it is true. Enter this in direct (immediate) mode to check out your computer:

```
PRINT 6>5, 5>7
```

The left hand expression is true, the right hand one false. So you'll get a −1 (or 1) and Ø displayed on the screen.

The integers you get as a result can be used in their own right for performing calculations within a program. Be careful with division though as you'll get an error message if you try to divide anything by Ø.

LOGICAL OPERATORS

The keywords AND, OR and NOT are logical operators which provide a powerful extension to IF ... THEN decision-making of the type seen on page 33. For example, IF this is true AND that is true, THEN do something. Certain other functions can be performed as well. These operators—sometimes called *Boolean* operators (don't worry about the long name)—can be used in direct or program modes to compare numbers or strings, and to obtain a 'truth value' of what may often be very complex test conditions. So they offer a shortcut alternative to what would otherwise be very involved conditional branches making messy use of IF ... THEN statements.

USING AND

The operator AND can, at its simplest, be considered to have the same meaning as in English. In an expression such as:

```
IF V>Ø AND V<1ØØ PRINT "IN RANGE"
```

the message is displayed only if V is greater than Ø *and* less than 1ØØ.

In a program, you may get a line like this:

```
990 OKAY=1 AND MNTH>5 AND
   MNTH<1Ø AND YEAR>19ØØ AND
   YEAR<2ØØ1
```

```
99Ø OKAY=−1 AND MNTH>5 AND
   MNTH<1Ø AND YEAR>19ØØ AND
   YEAR<2ØØ1
```

Such a program line could be used to check whether a date was in the summer in the twentieth century.

Here AND is used to conduct four tests, *each* of which must be true if OKAY is to remain true also. In this validation test the OKAY variable is first set to 'true' (−1, but 1 on the Sinclairs). Now, the requirement is that the MNTH falls in the range 6 to 9 and the YEAR between 19Ø1 and 2ØØØ. But let's look more closely. If all the conditions are met the expressions all yield a truth value. So, in effect, the program line converts to:

```
99Ø OKAY=−1 AND−1 AND −1 AND
   −1 AND−1
```

(1 rather than −1 on the Sinclairs)

by adding PRINT OKAY you can check that the result is indeed −1, or true.

USING OR

OR can be regarded in much the same way as AND although its meaning differs, as you might expect from its use in everyday English. Again, look at its use in a simple program line where it is used to compare the values of a variable:

```
IF V=8 OR V=1Ø PRINT "OKAY"
```

The message is displayed if the value of V is 8 *or* 1Ø.

OR is very useful when checking the validity of an INPUT to a program. If you're asked to enter your age, say, there is always someone who'll try out −1Ø or 999 just to see what happens. But you can get round the problem like this:

```
1Ø INPUT A
2Ø IF A<1 OR A>12Ø THEN PRINT
   "PLEASE BE SENSIBLE": GOTO 1Ø
```

USING NOT

The third commonly used logical operator is NOT. This differs slightly in that it acts *only* on the numerical or logical expression which follows it—AND and OR both compare expressions on either side, but NOT works only on a single value.

NOT can be used in a program line such as:

```
IF NOT (A>1Ø) THEN 999
```

In everyday English this could be translated into something like 'if the value of A is *not* greater than 1Ø, proceed to Line 999'.

The logical function of NOT is to convert a true condition into a false one—or vice versa. So you can use NOT as a switch or to reverse the true/false value which was the result of another test.

TRUTH TABLES

Quite often you'll come across references to 'truth tables' when mention is made of logical operators.

These are very simple and are used to give a visual representation of what occurs when true (−1 or 1) and false (Ø) values are ANDed or ORed. NOT doesn't really need a table as values are simply reversed.

Truth tables can take different forms. A typical one for AND is:

A	B	C	
−1	−1	−1	(line 1)
−1	Ø	Ø	(line 2)
Ø	−1	Ø	(line 3)
Ø	Ø	Ø	(line 4)

This doesn't mean much until you interpret line 1 as: 'If A is true and B is true then C is true'. The meaning of line 2 is: 'If A is true and B is untrue then C is untrue'. Line 3 means: 'If A is untrue and B is true, C is untrue'. Line 4 reads: 'If A and B are both untrue then C is untrue'. The truth table for OR is:

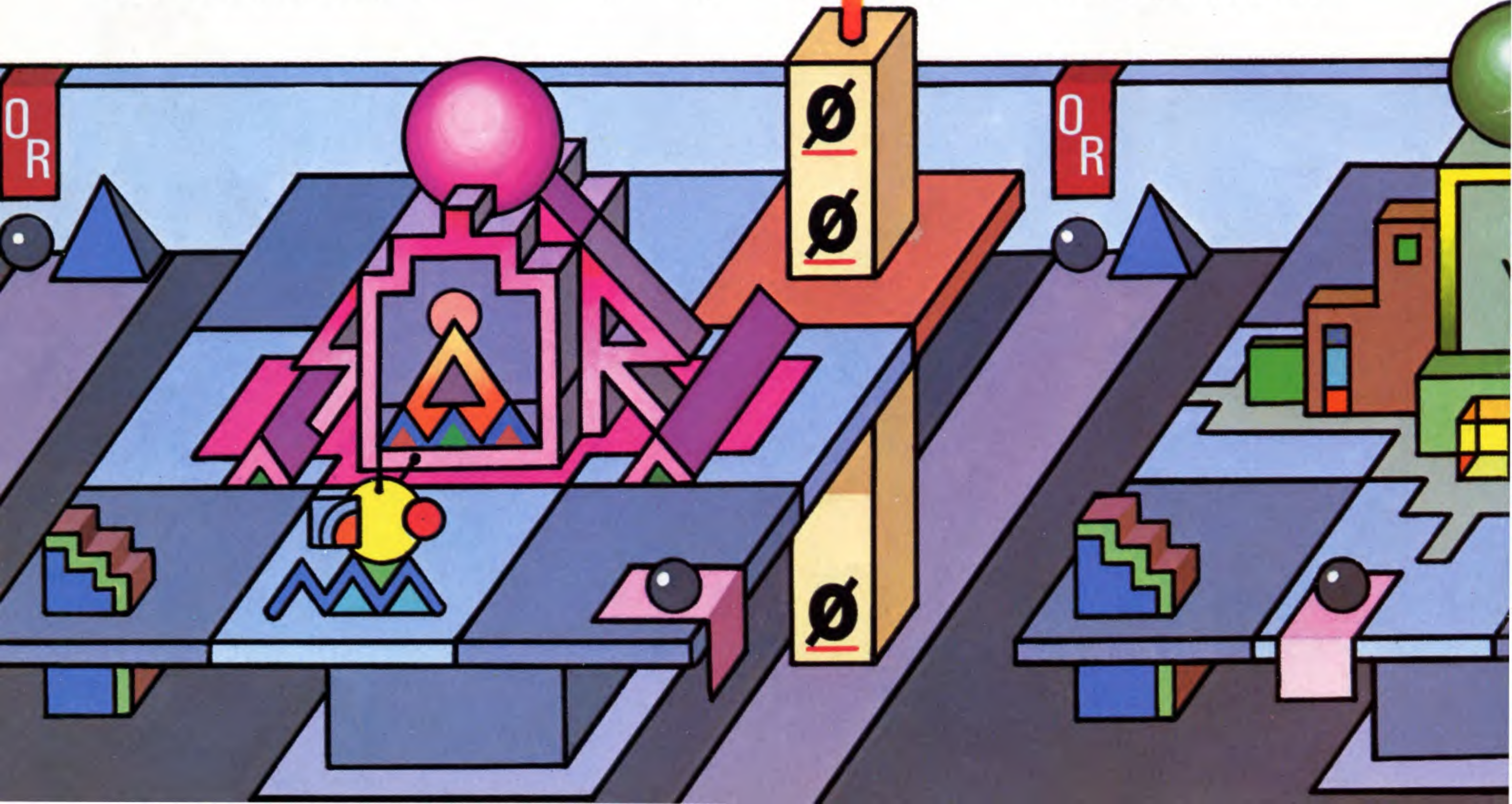
A	B	C
−1	−1	−1
−1	Ø	−1
Ø	−1	−1
Ø	Ø	Ø

The first line reads: 'If A is true and B is true then C is true'. Lines 2 and 3 can each be interpreted as: 'If either A or B is true then C is true'. Line 4 means: 'If A and B are both untrue then C also is untrue'.

IN PRACTICE

Finally, here is a program that uses AND, OR and NOT. It's a simplified version of a program that works out the correct salary scale for someone applying for a job:

```
1Ø INPUT "AGE"; AGE
2Ø INPUT "NUMBER OF O-LEVELS";
   OLEVELS
3Ø OVER18=(AGE>=18)
4Ø QUAL=(OLEVELS>=5)
5Ø IF NOT OVER18 AND NOT QUAL
   THEN PRINT "NOT SUITABLE"
6Ø IF(NOT OVER18 AND QUAL) OR(OVER18
   AND NOT QUAL) THEN PRINT"SALARY
   SCALE ONE"
7Ø IF OVER18 AND QUAL THEN
   PRINT "SALARY SCALE TWO"
```



Say you INPUT age 20 and 4 O-levels. This means that (AGE> =18) is true, but (OLEVELS> =5) is false. So the person is OVER18 and NOT QUALified. The computer soon finds this set of conditions in Line 6Ø and so it PRINTS out 'SALARY SCALE ONE'. You can easily extend this program to check more conditions—such as whether the person has more than two years experience, or anything else that's necessary for the job.

(Note, on some BBC computers you'll need to change the semicolons in Lines 1Ø and 2Ø to commas.)

USING EOR

The Acorn computers have one extra operator called EOR—short for Exclusive OR. It means you can have either one thing or another but not both. You might use it in an expression such as this:

```
IF buses< =2 EOR cars< =6 THEN PRINT
   "YOU CAN BOARD THE FERRY"
```

Using EOR prevents the ferry being overloaded with both buses and cars.

Here is the truth table for EOR:

A	B	C
−1	−1	Ø
−1	Ø	−1
Ø	−1	−1
Ø	Ø	Ø

Like the other operators, EOR can also be used numerically and it is particularly useful for switching back and forth between two numbers. If you have a line like:

```
S=S EOR 1
```

inside a loop and you set S equal to 1 at the start, it will equal Ø the first time round, 1 second time, Ø third time and so on. You could then use S to control various things within the program. For instance, COLOUR S would swap between COLOUR Ø and 1; PRINT CHR$ 224+S would swap between character 224 and 225—which is just what you might need for an animation sequence.

NUMERICAL OPERATIONS

Use of the logical operators is not confined to IF ... THEN statements as shown above. They can be used in certain types of numerical operations as well. You may have seen them used this way in programs and wondered what was happening. In fact they do not always work in the obvious way you might expect. Enter this in direct mode:

```
PRINT 375 AND 47
```

You might expect this to give a result of 422, or perhaps 37547. In fact, the result is 39. So what's happening? To explain this we have to delve slightly deeper into the working of the computer, to understand the way in which the two ANDed expressions in the second example are handled.

The first thing that happens is that the expressions (375 and 47) are converted into two-byte integers. Here's the binary form of both numbers:

Two-byte binary of 375:
ØØØØØØØ1Ø111Ø111
Two-byte binary of 47:
ØØØØØØØØØØ1Ø1111

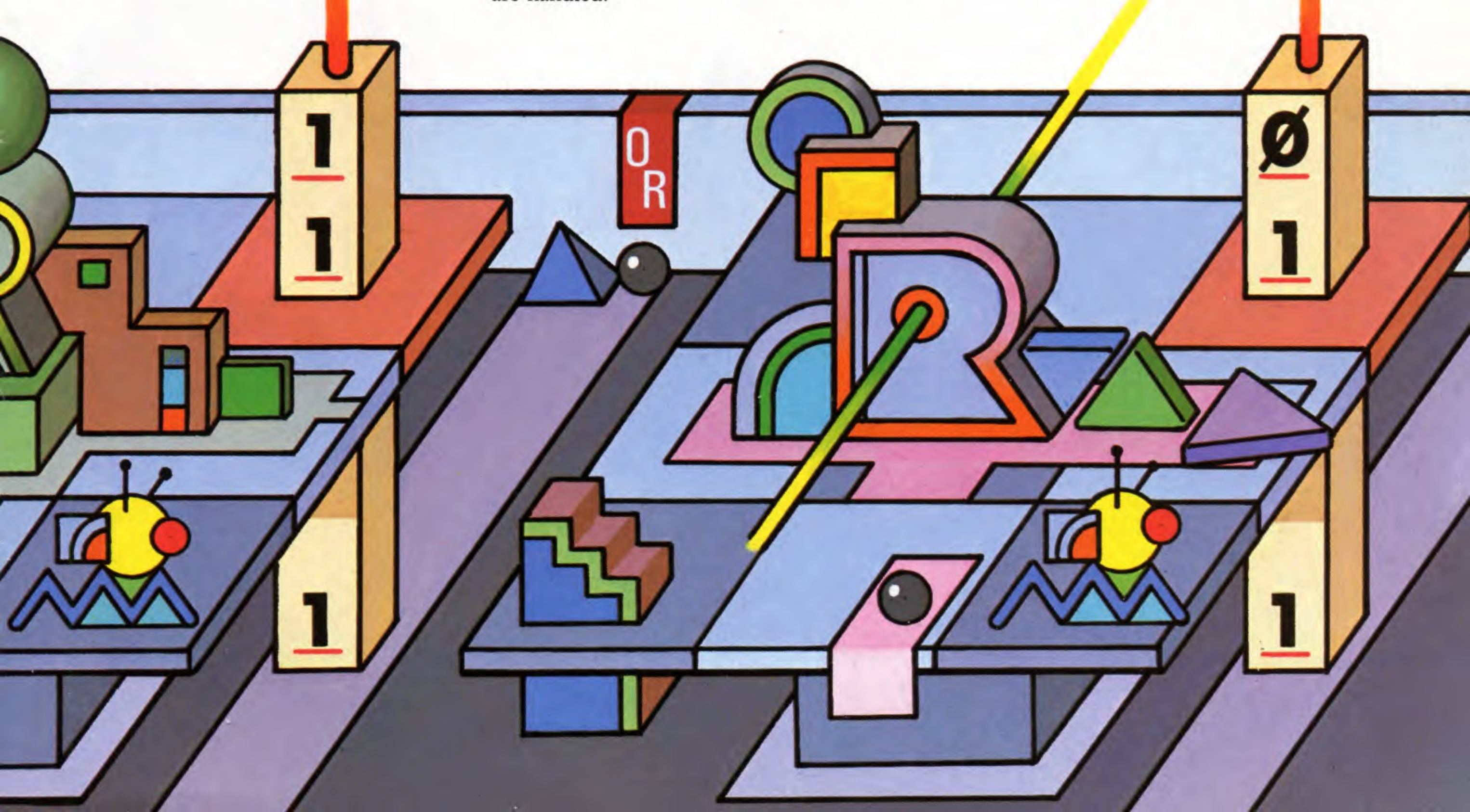
Now, AND does a bit-by-bit comparison of the 16 bits, returning 1 *only* when there's a 1 in the same bit of each binary number. Working from the right, only the zeroth, first, second and fifth bit-pairs fulfil this. This gives the binary number ØØØØØØØØØØ1ØØ111. Converting this back to decimal, bits 5, 2, 1 and Ø have a combined value of 39, the result of ANDing 375 with 47.

Now try this in direct mode:

```
PRINT 375 OR 47
```

So how is the result—383—obtained? The property of ORing is to give a bit result of 1 if *either one* of the bits in any bit-pair comparison is 1. Look at the two-byte binary form of the numbers again. Bits 8, 6, 5, 4, 3, 2, 1 and Ø all have a 1 in at least one part of the bit-pair. The resulting binary number is ØØØØØØØ1Ø1111111 which is 383 in decimal.

With OR it is possible to obtain the same bit value for different expressions. For example PRINT 375 OR 75 also yields the result 383. You can check this, if you wish, by converting the numbers to binary.



The value −1 (which, incidentally, is stored as FFFFFFFF in hex, a bit pattern all of 1s) is left unaltered when ORed with any valid number. To test this try in direct mode:

```
PRINT −1 OR 375
```

And the result is −1.

Now look at NOT in a numerical application

```
PRINT NOT 1Ø.75, NOT −11
```

This gives the results −11 and 1Ø. So NOT in effect adds 1 to the number being NOTed and then changes the sign. Note that the floating point number is rounded *down* to an integer and that the actual value can be estimated using X=−(X+1). In fact the value is converted into a four-byte integer whose bits are all reversed.

The logical operators on the Sinclair computers do not make bit-by-bit comparisons of numbers and so are rarely used in numerical operations.

BITWISE OPERATION

The logical operators have an additional function on Commodore computers as bit-by-bit operators capable of controlling the functions of the computer. On Commodore computers, certain memory locations perform a variety of tasks. You can specify a task by arranging a particular combination of bits to be held 'high'—to be switched on—at the same time suppressing others. The BASIC commands used to control the contents of any memory location are POKEs and PEEKs (see page 246), and a typical instruction may take the form:

```
POKE 1, PEEK (1) AND 251
```

This (on the Commodore 64) turns off bit 3. PEEK (1) gives 55 decimal, so let's see how this is ANDed:

```
Value 55:    ØØ11Ø111
Value 251:   11111Ø11
```

Now ANDing these bits gives: ØØ11ØØ11 (=51). Compare the old bit pattern of the value 55 with the bit pattern of 51 (the new PEEK value of location 1) and you can see that bit 3 (third from the right) has changed to Ø—effectively, it has been switched off. The situation can be reversed by using OR in this form:

```
POKE 1, PEEK (1) OR 4
```

The only 1 which appears in the binary form of 4 (ØØØØØ1ØØ) when ORed with the bit pattern of binary 51 (ØØ11ØØ11) gives ØØ11Ø111 which is the value of the PEEK before the previous operation was carried out—in other words it has reverted to 55.

All this may seem like a lot of trouble when in many instances you can actually use a direct POKE value rather than bother with logical operators. Here, though, there is a risk of accidentally readjusting other bits from their previous high or low setting. Using logical OR you need only specify the bit value(s) you wish turned on (that is set to 1): anything else that is already on is not affected.

Q+A

Is there any limit to the size of numbers used in AND and OR

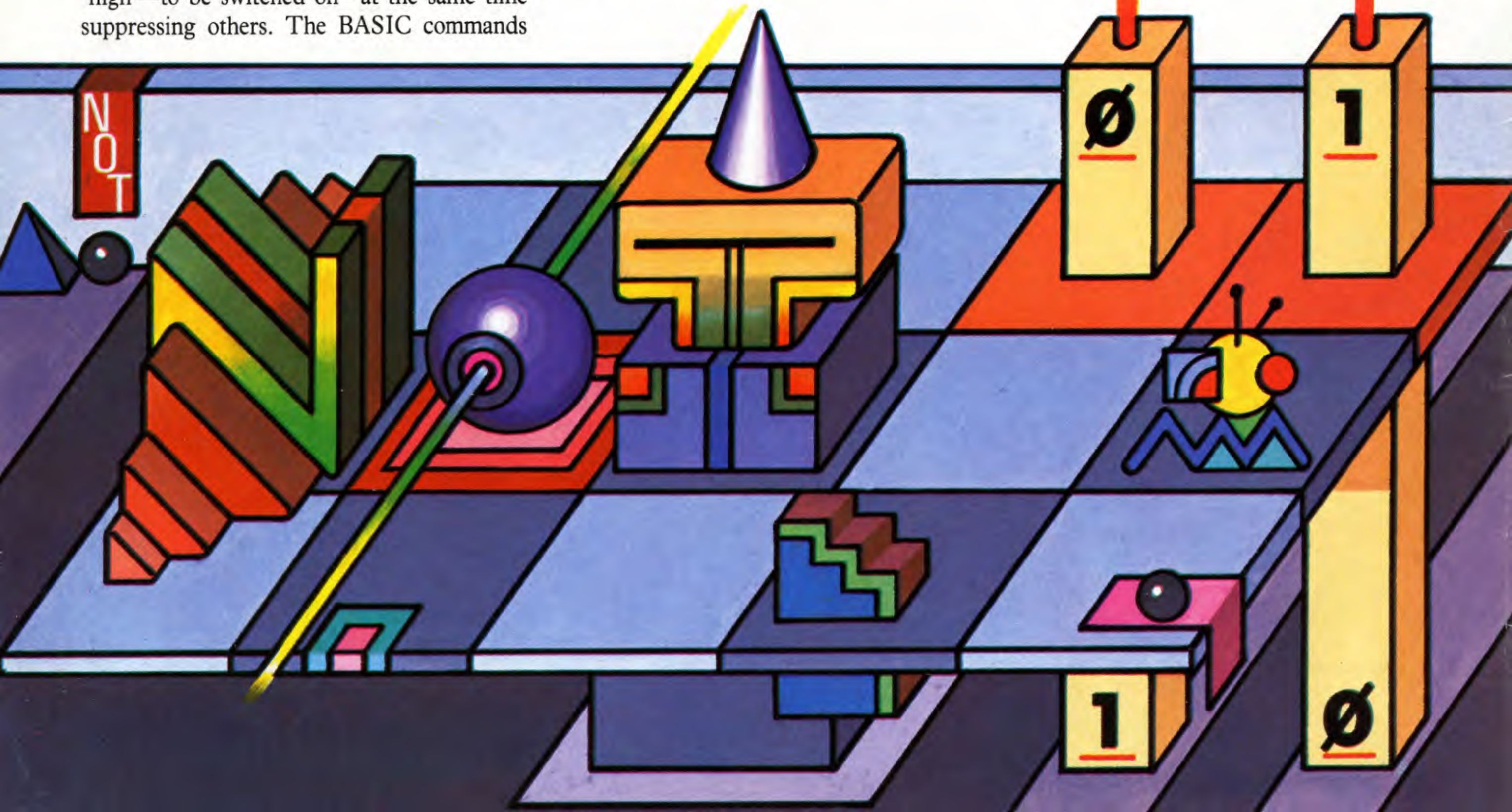
The Commodore 64, Vic 20, Dragon and Tandy computers cannot use numbers which exceed the range −32768 to +32767 in any ANDing or ORing operation. Exceeding this range produces an illegal quantity error message on the Commodores and a function call error message on the Dragon or Tandy.

The Acorn computers accept numbers of up to four bytes—minus $65536^2/2$ to plus $65536^2/2-1$—so any conceivable logical operation is likely to fall within range.

Numbers cannot be ANDed or ORed on the Spectrum.

AND can be used to switch off one or more bits simply by deducting the cumulative value of all remaining bits from 255. To switch off bits Ø and 2, for example, which is 5 in decimal, use AND 25Ø. To switch them back on you would use OR 5.

This type of procedure is sometimes referred to as *bit masking* and the difference from direct POKEing is that the latter changes bit values in accordance with the value you POKEd. Logical operators do not necessarily change anything but the bit you want changed.



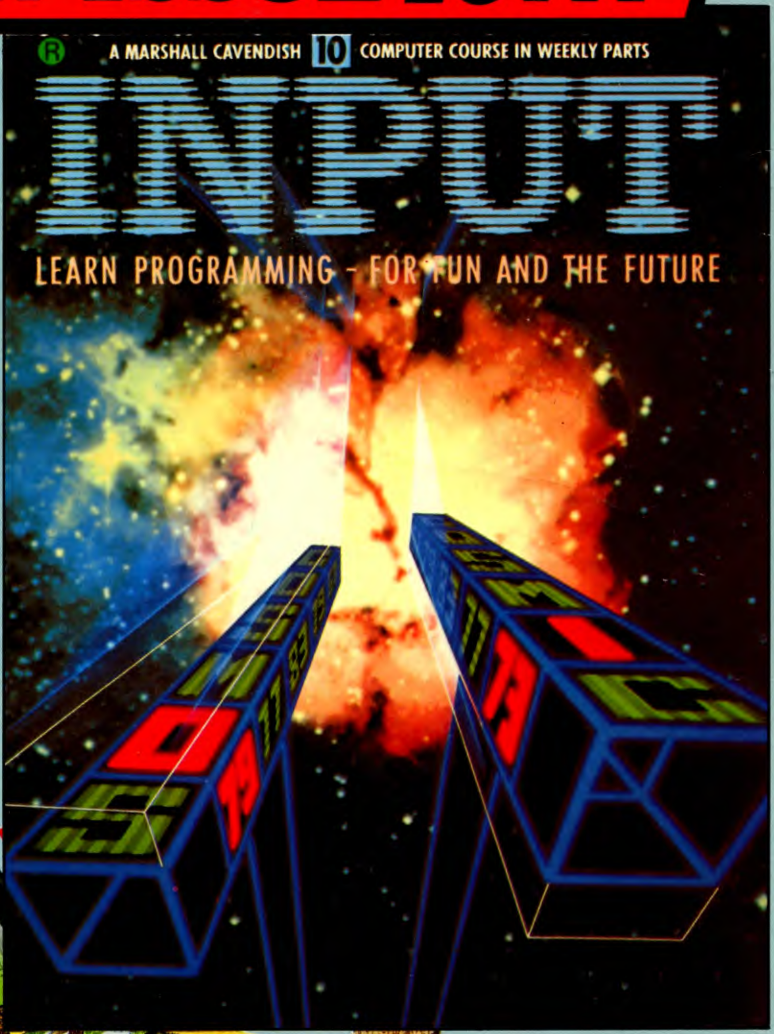
CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 10...

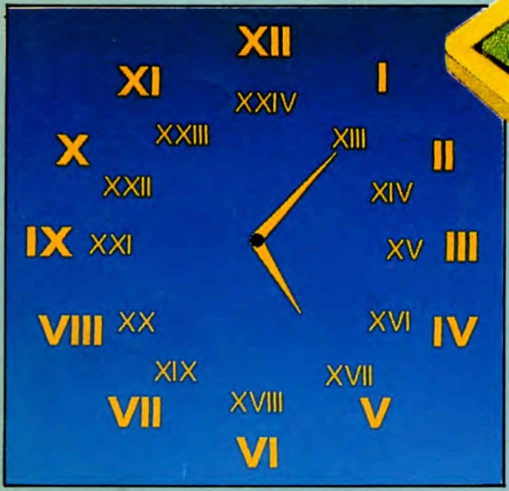
- ❑ *Follow the development of a complete adventure game. The first stage: MAPPING OUT THE ADVENTURE*
- ❑ *Learn about ASCII CODES—what they are, and how to make them work for you*
- ❑ *Fast, accurate typing is a valuable skill, whether you're writing a book or copying programs. Let your computer teach you with our TYPING TUTOR*
- ❑ *Get started on ASSEMBLY LANGUAGE, and learn how to translate it into machine code*
- ❑ *There's more on SIN and COS, the maths functions that you can use for all kinds of fascinating graphic effects*

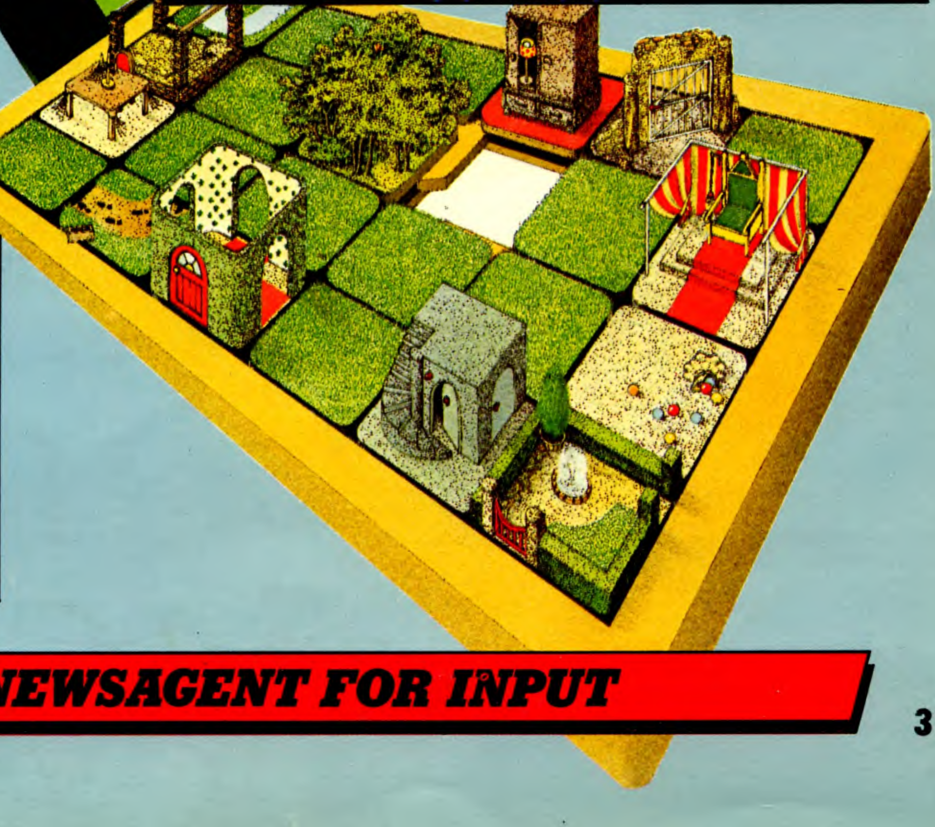


A MARSHALL CAVENDISH 10 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE





ASK YOUR NEWSAGENT FOR INPUT